Journal of Computer Security 0 (0) 1
IOS Press

An interpolation-based method for the verification of security protocols[1]

Marco Rocchetto [a], Luca Viganò [b] and Marco Volpe [c]
[a] *ALES, United Technologies Research Center, Italy*
marco.rocchetto@utrc.utc.com
[b] *Department of Informatics, King's College London, UK*
luca.vigano@kcl.ac.uk
[c] *INRIA and LIX, École Polytechnique, France*
marco.volpe@inria.fr

Abstract. Interpolation has been successfully applied in formal methods for model checking and test-case generation for sequential programs. Security protocols, however, exhibit idiosyncrasies that make them unsuitable for the direct application of interpolation. We address this problem and present an interpolation-based method for security protocol verification. Our method starts from a protocol specification and combines Craig interpolation, symbolic execution and the standard Dolev-Yao intruder model to search for possible attacks on the protocol. Interpolants are generated as a response to search failure in order to prune possible useless traces and speed up the exploration. We illustrate our method by means of concrete examples and discuss the results obtained by using a prototype implementation.

Keywords: Security protocols, Craig interpolation, Symbolic execution, Verification

1. Introduction

A number of tools (e.g., [1–8] just to name a few) have been developed for the analysis of security protocols at *design time*: starting from a formal specification of a protocol and of a security property it should achieve, these tools typically carry out model checking or automated reasoning to either *falsify* the protocol (i.e., find an attack with respect to that property) or, when possible, *verify* it (i.e., prove that it does indeed guarantee that property, perhaps under some assumptions such as a bounded number of interleaved protocol sessions [9]). While verification is, of course, the optimal result, falsification is also extremely useful as one can often employ the discovered attack trace to directly carry out an attack on the protocol implementation (e.g., [10]) or exploit the trace to devise a suite of test cases so as to be able to analyze the implementation at *run-time* (e.g., [11–13]).

Such an endeavor has already been undertaken in the programming languages community, where, for instance, *interpolation* has been successfully applied in formal methods for model checking and test-case generation for sequential programs, e.g., [14–17], with the aim of reducing the dimensions

[1]Work partially supported by the FP7-ICT-2009-5 Project no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services" and the PRIN 2010-11 project "Security Horizons". Much of this work was carried out while the authors were at the Dipartimento di Informatica, Università di Verona, Italy, and while Marco Rocchetto was at iTrust at the Singapore University of Technology and Design. We thank Giacomo Dalle Vedove, Marco Palamà and Fabio Pettenuzzo.

of the search space. Since a state space explosion often occurs in security protocol verification, we expect interpolation to be useful also in this context. Security protocols, however, exhibit idiosyncrasies that make them unsuitable for the direct application of the standard interpolation-based methods, most notably, the fact that the presence of a Dolev-Yao intruder [18] gives a security protocol a flavor of non-determinism, makes it a non-sequential program (since the intruder, who is in complete control of the network, can freely interleave his actions with the normal protocol execution) and requires taking care of the deduction capabilities of the intruder.

In this paper, we address this problem and present an interpolation-based method for security protocol verification. Our method starts from the formal specification of a protocol and of a security property and combines Craig interpolation [19], symbolic execution [20] and the standard Dolev-Yao intruder model [18] to search for goals (representing attacks on the protocol). Interpolation is used to prune possible useless traces and speed up the exploration. More specifically, our method proceeds as follows: starting from a specification of the input system, including protocol, property to be checked and a finite number of session instances (possibly generated automatically by using a preprocessor), it first creates a corresponding sequential non-deterministic program, according to a procedure that we have devised, and then defines a set of goals and searches for them by symbolically executing the program. When a goal is reached, an attack trace can be extracted from the constraints that the execution of the path has produced; such constraints represent conditions over parameters that allow one to reconstruct the attack trace found. When the search fails to reach a goal, a backtrack phase starts, during which the nodes of the graph are annotated (according to an adaptation of the algorithm defined in [15] for sequential programs) with formulas obtained by using Craig interpolation. Such formulas express conditions over the program variables, which, when implied from the program state of a given execution, ensure that no goal will be reached by going forward and thus that we can discard the current branch. The output of the method is a proof of (bounded) correctness in the case when no goal location can be reached; otherwise all the discovered (one or more) attack traces are produced.

In order to show that our method concretely speeds up the validation, we have implemented a Java prototype called *SPiM* (*Security Protocol interpolation Method*). We report here also on some experiments that we have performed: we considered seven case studies and compared the analysis of SPiM with and without interpolation, thereby showing that interpolation does indeed speed up security protocol verification by reducing the search space and the execution time. We also compare the SPiM tool with the three state-of-the-art model checkers for security protocols that are part of the AVANTSSAR platform [1], namely, CL-AtSe [21], OFMC [3] and SATMC [22]. This comparison shows, as we expected, that SPiM is not yet as efficient as these mature tools but that there is considerable room for improvement, e.g., by enhancing our interpolation-based method with some of the optimization techniques that are integrated in the other tools.

Summarizing, we list the contributions of this work as follows.

(1) We define a translation of security protocols into sequential programs and we prove the correctness of this translation.

(2) By adapting existing program analysis techniques, we propose a new approach for security protocol verification that combines Craig interpolation, symbolic execution and the standard Dolev-Yao intruder.

(3) We implement our technique in a tool called SPiM and we show that Craig interpolation produces a speed-up in the verification process up to 70%.

We proceed as follows. In Section 2, we provide some (fairly standard) background on security protocol verification, discussing the algebra of protocol messages, the Dolev-Yao intruder, the two security protocol specification languages ASLan++ and ASLan that we consider in our method (which is however open to the integration with other protocol specification languages), and the running example (the NSL protocol) that we will consider in the rest of the paper. In Section 3, we introduce SiL, the input language of our SPiM tool, which is a simple imperative programming language that we use to define the sequential programs to be analyzed by the verification algorithm. We also give the details of the translation procedure from security protocols into sequential programs, for one and more protocol sessions, and prove the correctness of the translation (i.e., that it does not introduce nor delete attacks with respect to the input ASLan++ specification). In Section 4, we present our interpolation algorithm, which is a slightly simplified version of McMillan's IntraLA algorithm [15], and show it at work for our running example. In Section 5, we introduce the SPiM tool, discuss the experiments that we have performed and describe the interpolants generated by the tool during the analysis. In Section 6, we discuss further related work (in addition to the works already considered in the rest of the paper), and we conclude in Section 7 by summarizing our main results and discussing future work. Additional details (examples and a proof of one of the lemmas) are given in appendix. This paper extends and supersedes [23].

2. Background

We provide some (fairly standard) background on security protocol verification and briefly describe the two specification languages ASLan++ and ASLan.

2.1. Messages

Security protocols describe how agents exchange messages, built using cryptographic primitives, in order to obtain security guarantees such as confidentiality or authentication. Protocol specifications are parametric and prescribe a general recipe for communication that can be used by different agents playing in the protocol roles (sender, receiver, server, etc.). The *algebra of messages* tells us how messages are constructed. Following standard practice (e.g., [3, 24]), we consider a countable *signature* Σ and a countable set *Var* of *variable symbols* disjoint from Σ, and write Σ^n for the symbols of Σ with arity n; thus Σ^0 is the set of *constants*, which we assume to have disjoint subsets that we refer to as *agent names* (or just *agents*), *public keys*, *private keys*, *symmetric keys* and *nonces*. The variables are, however, untyped (unless denoted otherwise) and can be instantiated with arbitrary types, yielding an *untyped model*. We will use upper-case letters to denote variables (e.g., $A, B, \ldots$ for agents, N for nonces, etc.) and lower-case letters to denote the corresponding constants (concrete agents names, concrete nonces, etc.). All these may be possibly annotated with subscripts and superscripts.

The symbols of Σ that have arity greater than zero are partitioned into the set Σ_p of *(public) operations* and the set Σ_m of *mappings*. The public operations represent all those operations that every agent (including the intruder) can perform on messages they know. In this paper, we consider the following public operations:[2]

- $\{M_1\}_{M_2}$ represents the *asymmetric encryption* of M_1 with public key M_2;

[2]We could, of course, quite straightforwardly add other operations, e.g., for hash functions, but refrain from doing so for the sake of simplicity.

- $\{M_1\}_{inv(M_2)}$ represents the *asymmetric encryption* of M_1 with private key $inv(M_2)$ (the mapping $inv(\cdot)$ is discussed below);
- $\{|M_1|\}_{M_2}$ represents the symmetric encryption of M_1 with symmetric key M_2;
- $[M_1, M_2]$ (or simply M_1, M_2 when there is no risk of confusion) represents the concatenation of M_1 and M_2.

In contrast to the public operations, the mappings of Σ_m are those functions that do not correspond to operations that agents can perform on messages, but that map between constants. In this paper, we use the following two mappings. First, $inv(M)$ gives the private key that corresponds to the public key M. Second, for long-term key infrastructures, we assume that every agent A has a public key $pk(A)$ and a corresponding private key $inv(pk(A))$; thus $pk(\ldots)$ is a mapping from agents to public keys. In the same way, one may model further long-term key infrastructures, e.g., using $sk(A,B)$ to denote a shared key of agents A and B.

Since the mappings map from constants to constants, we consider a term like $inv(pk(a))$ as atomic as its construction does not involve any operation performed by an honest agent or the intruder, nor is there a way to "decompose" such a message into smaller parts. Since we will also deal with terms that contain variables, let us call *atomic* all terms that are built from constants in Σ^0, variables in *Var*, and the mappings of Σ_m. The set $\mathcal{T}_\Sigma(Var)$ of all *terms* is the closure of the atomic terms under the operations of Σ_p. A *ground term* is a term without variables, and we denote the set of ground terms with $\mathcal{T}_\Sigma$.

As is often done in security protocol verification, we interpret terms in the *free algebra*, i.e., every term is interpreted by itself and thus two terms are equal iff they are syntactically equal (e.g., two constant symbols n_1 and n_2 immediately represent different values). Numerous algebras have been considered in security protocol verification, e.g. [25, 26], ranging from the free algebra to various formalizations of algebraic properties of the cryptographic operators employed. Here, for simplicity, we consider only the free algebra in order to be able to focus on the introduction of our interpolation method. Moreover, our results require a bound on the message depth (that we introduce later, in Section 4.3), but, fortunately, such a bound is known for the free algebra when considering a finite number of sessions (see, e.g., [9]). We believe that, in principle, our interpolation method could be applied to more complex algebras (e.g., for protocols that make use of modular exponentiation or xor) as long as such a bound can be established for the considered equational theory. We leave this investigation for future work.

2.2. The Dolev-Yao Intruder

For concreteness and brevity, we consider here the standard Dolev and Yao [18] model of an active intruder, denoted by i, who controls the network but cannot break cryptography; note, however, that our approach is independent of the actual strength of the intruder and weaker (or stronger, e.g., being able to attack the cryptography) intruder models could be considered.

The intruder i can intercept messages and analyze them if he possesses the corresponding keys for decryption, and he can generate messages from his knowledge and send them under any agent name. For a set IK of messages, we define $DY(IK)$ (for "Dolev-Yao" and "Intruder Knowledge") to be the smallest set closed under the standard *generation (G)* and *analysis (A) rules* of the *system* $\mathcal{N}_{DY}$ given in Fig. 1. The G rules express that the intruder can compose messages from known messages using pairing, asymmetric and symmetric encryption. The A rules describe how the intruder can decompose messages.

$$\frac{M \in IK}{M \in DY(IK)}\ G_{\text{axiom}} \qquad \frac{M_1 \in DY(IK) \quad M_2 \in DY(IK)}{[M_1, M_2] \in DY(IK)}\ G_{\text{pair}}$$

$$\frac{M_1 \in DY(IK) \quad M_2 \in DY(IK)}{\{M_1\}_{M_2} \in DY(IK)}\ G_{\text{crypt}} \qquad \frac{M_1 \in DY(IK) \quad M_2 \in DY(IK)}{\{|M_1|\}_{M_2} \in DY(IK)}\ G_{\text{scrypt}}$$

$$\frac{[M_1, M_2] \in DY(IK)}{M_i \in DY(IK)}\ A_{\text{pair}_i} \qquad \frac{\{|M_1|\}_{M_2} \in DY(IK) \quad M_2 \in DY(IK)}{M_1 \in DY(IK)}\ A_{\text{scrypt}}$$

$$\frac{\{M_1\}_{M_2} \in DY(IK) \quad inv(M_2) \in DY(IK)}{M_1 \in DY(IK)}\ A_{\text{crypt}} \qquad \frac{\{M_1\}_{inv(M_2)} \in DY(IK) \quad M_2 \in DY(IK)}{M_1 \in DY(IK)}\ A^{-1}_{\text{crypt}}$$

Fig. 1. The system $\mathcal{N}_{DY}$ of rules of the Dolev-Yao intruder.

2.3. ASLan++ and ASLan

We give here a brief overview of the security protocol specification languages ASLan++ [27] and ASLan [28], focusing on the aspects relevant to our method. We remark that our methodology can be easily adapted to work with other protocol specification languages (which, like ASLan++, typically specify the different protocol roles as interacting processes) by providing a translator to the SiL input language as described in Section 3.2.

ASLan++ is a formal and typed security protocol specification language, whose semantics is defined in terms of the more low-level language ASLan, which we describe below.

Hierarchy of entities. An ASLan++ specification consists of a hierarchy of *entity declarations*, which are similar to Java classes. The top-level entity is usually called *Environment* (similar to the "main" procedure of a program) and it typically contains the definition of a *Session* entity, which in turn contains a number of sub-entities representing all the parties involved in a protocol. Each entity of an ASLan++ specification is composed of two main sections: *symbols*, in which there is the instantiation of all the variables and constants used in the entity, and *body*, in which the behavior of the entity is described (e.g., message exchange).

The body of an entity. Inside the body of an entity we use three different types of statements: assignment, message send and message receive. An *assignment* has the form `Var := constant`, which assigns to the variable `Var` a constant of the proper type (a new constant is generated if `Var := fresh()` is used). A *message send* statement, `Sender -> Receiver: M`, is composed of two variables `Sender` and `Receiver` representing sender and receiver, respectively, and a message `M` exchanged between the two parties. In *message receive*, `Sender` and `Receiver` are swapped and usually, in order to assign a value to the variable `M`, a `?` precedes the message `M`, i.e., `Sender -> Receiver: ?M`. In ASLan++, the **`Actor`** keyword refers to the entity itself (similar to "this" or "self" in object-oriented languages) and thus we actually write the send and receive statements as **`Actor`** `-> Receiver: M` and `Sender ->` **`Actor`**`: ?M`, respectively.

$A \to i : \{N_A, A\}_{pk(i)}$	$Alice_1.Actor \to Alice_1.B : \{Alice_1.Na, Alice_1.Actor\}_{pk(Alice_1.B)}$	$a \to i : \{c_1, a\}_{pk(i)}$
$i(A) \to B : \{N_A, A\}_{pk(B)}$	$? \to Bob_2.Actor : \{Bob_2.Na, Bob_2.A\}_{pk(Bob_2.Actor)}$	$i(a) \to b : \{c_1, a\}_{pk(b)}$
$B \to i(A) : \{N_A, N_B\}_{pk(A)}$	$Bob_2.Actor \to Bob_2.A : \{Bob_2.Na, Bob_2.Nb\}_{pk(Bob_2.A)}$	$b \to i(a) : \{c_1, c_2\}_{pk(a)}$
$i \to A : \{N_A, N_B\}_{pk(A)}$	$Alice_1.B \to Alice_1.Actor : \{Alice_1.Na, Alice_1.Nb\}_{pk(Alice_1.Actor)}$	$i \to a : \{c_1, c_2\}_{pk(a)}$
$A \to i : \{N_B\}_{pk(i)}$	$Alice_1.Actor \to Alice_1.B : \{Alice_1.Nb\}_{pk(Alice_1.B)}$	$a \to i : \{c_2\}_{pk(i)}$
$i(A) \to B : \{N_B\}_{pk(B)}$	$Bob_2.A \to Bob_2.Actor : \{Bob_2.Nb\}_{pk(Bob_2.Actor)}$	$i(a) \to b : \{c_2\}_{pk(b)}$

Fig. 2. Man-in-the-middle attack on the NSPK protocol (left), symbolic attack trace at state 15 of the algorithm execution (middle) and instantiated attack trace obtained with our method (right).

```
1 entity Alice(Actor, B: agent) {
2  symbols
3   Na, Nb: text;
4  body{
5   Na := fresh();
6   Actor -> B: {Na,Actor}_pk(B);
7   B -> Actor: {Na,?Nb,B}_pk(Actor);
8   Actor -> B: {auth:(Nb)}_pk(B);
9  }
10 }
```

```
1 entity Bob(A, Actor: agent) {
2  symbols
3    Na, Nb: text;
4  body{
5   ? -> Actor: {?Na,?A}_pk(Actor);
6   Nb := fresh();
7   Actor -> A: {Na,Nb,Actor}_pk(A);
8   A -> Actor: {auth:(Nb)}_pk(Actor);
9  }
10 }
```

Fig. 3. Partial ASLan++ specification for the protocol NSL.

Example 1. *As a running example, we will use NSL, the Needham-Schroeder Public Key (NSPK) protocol with Lowe's fix [7], which aims at mutual authentication between A and B:*

$$A \to B : \{N_A, A\}_{pk(B)}$$
$$B \to A : \{N_A, N_B, B\}_{pk(A)}$$
$$A \to B : \{N_B\}_{pk(B)}$$

The presence of B in the second message prevents the man-in-the-middle attack that NSPK suffers from, which is shown on the left of Fig. 2, where we write $i(A)$ to denote that the intruder is impersonating the honest agent A (that is, $i(x)$ denotes the intruder playing the role of x, for x an agent name.)

We give the complete ASLan++ specification for the protocol NSL in Appendix A. In Figure 3, we briefly describe only the section modeling the behavior of the two entities involved. Note that, for readability, from now on, we use math fonts instead of mixing math and typewriter fonts (e.g., we write $iknows(Payload)$ instead of `iknows(Payload)`*) in the text, while we use typewriter in code listings.*

The two roles are Alice, who is the initiator *of the protocol, and Bob, the* responder. *The elements between parentheses in line 1 declare which variables are used to denote the agents playing the different roles. Along the specification of the role Alice: Actor refers to the agent playing the role of Alice itself, while B is the variable referring to the agent who Alice believes is playing the role of Bob. Similarly, the section symbols declares that Na and Nb are variables of type* text*, which is the type used in ASLan++*

for arbitrary messages. The section body specifies the behavior of the role. First, the operation fresh() *assigns to the nonce Na a value that is different from the value assigned to any other nonce. Then Alice sends the nonce, together with her name, to the agent B, encrypted with B's public key. In line 7, Alice receives her nonce back together with a further variable (expected to represent B's nonce in a regular session of the protocol) and the name of B, all encrypted with her own public key. As a last step, Alice sends to B the nonce Nb encrypted with B's public key.*

The variable declarations and the behavior of Bob are specified by the listing on the right. We omit a full description of the code and only remark that the "?*" in the beginning of line* 5 *denotes the fact that the sender of such a message can be any agent, though no assignment is made for* ? *in that case.* □

Description of goals. Finally, we describe here two kinds of protocol goals in ASLan++. A *channel goal*, `label(_): Sender <chn> Receiver;`, defines a property `<chn>` that holds on all (the "_" is a wildcard) the exchanged messages labeled with `label` between the two entities `Sender` and `Receiver`. Labels are used to specify the class of messages for which a given property is required to be satisfied. For example, we use *authentication goals* defined as `auth_goal(_): Sender *-> Receiver;`, where `*->` specifies the fact that the receiver authenticates the sender. A *secrecy goal* is defined with `label(_): {Sender, Receiver}`, which states that each message labeled with `label` can only be shared between the two entities `Sender` and `Receiver`.

Example 2. *In the NSL running example, we want to verify whether the man-in-the-middle attack known for the NSPK protocol can be still applied after Lowe's fix. The scenario we are interested in can be obtained by the following ASLan++ instantiation:*

```
1  body { % of Environment
2    any Session(a,i);
3    any Session(a,b);
4  }
```

In session 1*, the roles of Alice and Bob are played by the agents a and i, respectively, whereas in session* 2 *they are played by a and b.*

A set of goals needs also to be specified. For simplicity, here we only require to check the authentication property with respect to the nonce of Bob, i.e., we will verify that the responder Bob authenticates the initiator Alice.

```
1  goals { auth:(_) A *-> B; }
```

□

Translation from ASLan++ into ASLan. As discussed in [1], an ASLan++ specification can be automatically translated into a more low-level ASLan specification, which ultimately defines a *transition system* $M = \langle \mathbf{S}, \mathbf{I}, \rightarrow \rangle$, where $\mathbf{S}$ is the *set of states*, $\mathbf{I} \subseteq \mathbf{S}$ is the set of initial states, and $\rightarrow \subseteq \mathbf{S} \times \mathbf{S}$ is the *(reflexive) transition relation*. A state is defined as a set of ground facts, i.e., the set of predicates holding in that state, all other ground facts being false (closed-world assumption). The structure of an ASLan specification is composed of six different sections: signature of the predicates, types of variables and constants, initial state, Horn clauses, transition rules of $\rightarrow$ and protocol goals. The content of the sections is intuitively described by their names. In particular, an initial state $I \in \mathbf{I}$ is composed of the concatenation of all the predicates that hold before applying any rewrite rule (e.g., the agent names and the intruder's own keys).

The specifications that we consider in this paper do not use Horn clauses, but rather a so called *Prelude* file, in which all the actions of the DY intruder are defined as a set H of Horn clauses, is automatically imported during the translation from ASLan++ into ASLan (see [28]).

The transition relation $\rightarrow$ is defined as follows. For all $S \in \mathbf{S}$, $S \rightarrow S'$ iff there exist

- a rule such that

$$PP.NP\&PC\&NC \; \exists [V] \Rightarrow R\,,$$

  where PP and NP are sets of positive and negative predicates, PC and NC conjunctions of positive and negative atomic conditions, and
- a substitution $\sigma : \{v_1, \ldots, v_n\} \rightarrow \mathcal{T}_\Sigma$, where $v_1, \ldots, v_n$ are the variables that occur in PP and PC such that:
  (1) $PP\sigma \subseteq \lceil S \rceil^H$, where $\lceil S \rceil^H$ is the closure of S with respect to the set of clauses H,
  (2) $PC\sigma$ holds,
  (3) $NP\sigma\sigma' \cap \lceil S \rceil^H = \emptyset$ for all substitutions σ' such that $NP\sigma\sigma'$ is ground,
  (4) $NC\sigma\sigma'$ holds for all substitutions σ' such that $NC\sigma\sigma'$ is ground and
  (5) $S' = (S \setminus PP\sigma) \cup R\sigma\sigma''$, where σ'' is any substitution such that for all $v \in V$, $v\sigma''$ does not occur in S.

We now define the translation of the relevant ASLan++ constructs to ASLan. Every ASLan++ entity is translated into a new *state predicate* and added to the section signature. This predicate is parametrized with respect to a *step label* (that uniquely identifies every instance) and it mainly keeps track of the local state of an instance (current values of whose variables) and expresses the control flow of the entity by means of step labels. As an example, if we have the ASLan++ entity

```
1 entity Sender(Actor, Receiver: agent){
2  symbols
3     Var: message;
4 }
```

the predicate `state_Sender` is added to the section signature and, assuming an instantiation of the entity **`new`** `Sender(sender, receiver)`, the new predicate

```
1 state_Sender(sender, iid, sl_0, receiver, dummy_message)
```

is used in transition rules to store all the informations of an entity, where the ID `iid` identifies a particular instance, `sl_0` is the step label, the parameters **`Actor`**, `Receiver` are replaced with constants `sender` and `receiver`, respectively, and the message variable `Var` is initially instantiated with `dummy_message`.

Given that an ASLan++ specification is a hierarchy of entities, when an entity is translated into ASLan, this hierarchy is preserved by a `child(id_1, id_0)` predicate that states that `id_0` is the parent entity of `id_1` and both `id_0` and `id_1` are entity IDs.

A variable assignment statement is translated into a transition rule inside the rules section. As an example, if in the body of the entity `Sender` defined above there is an assignment `Var := constant ;`, where `constant` is of the same type of `Var`, then we obtain the following transition rule:

```
1 state_Sender(Actor,IID,sl,Receiver,Var)
2 =>
3 state_Sender(Actor,IID,succ(sl),Receiver,constant)
```

which, for a given entity specified by *IID*, produces a predicate where the step label is increased and the variable is replaced by a constant. In the case of assignments to `fresh()`, the variable `Var` is assigned to a new variable.

In the case of a message exchange (sending or receiving statements) the `iknows(message)` predicate is added to the left-hand side of the corresponding ASLan rule. This states that the message `message` has been sent over the network, where `iknows` stands for *intruder knows* and is used because, as is usual, the Dolev-Yao intruder is identified with the network itself.

The last point we discuss is the translation of goals focusing on authentication and secrecy described above. The label in a send statement (e.g., **`Actor`** `-> Receiver: auth:(Na)`) generates a new predicate `witness(`**`Actor`**`,Receiver,label,Payload)` that is inserted into the ASLan transition rule representing the send statement. An equivalent `request(`**`Actor`**`,Sender,label,Payload,IID)` predicate is added for receive statements. These predicates are used in the translation of goals. In fact, an authentication goal is translated into the state (i.e., attack state)

```
1 not(dishonest(Sender)).
2 not(witness(Sender, Receiver, auth, Payload)).
3 request(Receiver, Sender, auth, Payload, IID)
```

where `not(dishonest(Sender))` states the sender `Sender` must not be the intruder, `not(witness(Sender, Receiver, auth, Payload))` states the payload of the authentication message must not be sent by the honest agent `Sender` and the last `request` predicate states the receiver `Receiver` has received the authentication message. A secrecy goal is translated into the attack state

```
1 iknows(Payload).
2 not(contains(i, Knowers)).
3 secret(Payload, label, Knowers)
```

where `iknows(Payload)` means that the intruder knows the payload, that the set of knowers (`Sender` and `Receiver` in the example above) does not contain the intruder `i` and the `secret` predicate is used to check the goal only when the rule containing the secrecy goal label is fired. This is because a `secret(Payload, label, Knowers)` predicate is added to all the transition rules that are translations of statements in which the payload of the secrecy goal is used. The declaration of an attack state `AS` amounts to adding a rule `AS => AS.attack` for a nullary predicate `attack`.

Example 3. *With regard to the NSL example, we show the ASLan specification corresponding to the translation of line* 7 *of the entity Alice (Figure 3):*

```
1 iknows(crypt(pk(E_S_A_Actor), pair(Na, pair(Nb_1, E_S_A_B)))).
2 state_Alice(E_S_A_Actor, E_S_A_IID, 3, E_S_A_B, Na, Nb)
3 =>
4 state_Alice(E_S_A_Actor, E_S_A_IID, 4, E_S_A_B, Na, Nb_1)
```

The original ASLan++ statement is a receive action. Its translation corresponds to (i) adding a new iknows predicate, concerning the message received and (ii) updating the state fact of Alice: in particular, in the transition rule, the step label is incremented (from 3 *to* 4*) and the argument referring to the variable Nb, which is the only one preceded by* ? *in the ASLan++ specification, gets the value of the nonce contained in the message.* □

3. Translating security protocols into sequential programs

3.1. The SPiM Input Language SiL

In Fig. 4, we present the full grammar of the SPiM Input Language SiL, a simple imperative programming language that we will use to define the sequential programs to be analyzed by the verification algorithm.

Definition 1. The *SPiM Input Language SiL* is defined by the grammar in Fig. 4, where X ranges over a set of *variable locations Loc* and c ranges over the set $\Sigma^0 \cup \mathbb{N}$. □

The basic *terms* of the language, constants and variable locations, are in the syntactic category E. A *message* M is a constant, a variable location, a concatenated message or some form of encrypted message.

The category L denotes *lists of messages*, whereas S stands for a *set of messages*: here IK is a special identifier referring to the intruder knowledge and $+$ is used to denote the union operation between sets.

B denotes the class of *Booleans*. In addition to the standard Boolean constants and operators, SiL contains two specific predicates: $IK \vdash M$, which intuitively evaluates to true when the message M is derivable from the set of messages in IK, and *witness*, with three arguments (a sender, a receiver, and a message), which is used in order to verify an authentication goal.

Finally, the *statements* of SiL, in the category C, comprise standard constructs (like assignments, conditionals and concatenation) together with mechanisms used to handle specific aspects of security protocols, like the possibility of setting the values of the message set variable IK, the ternary predicate *witness* and the boolean variable *attack*. The latter is set to *true* when an attack is found.[3]

Definition 2. We denote with $V = Loc \cup \{IK, attack, witness\}$ the set of *program variables* and with $D = \Sigma^0 \cup \mathbb{N} \cup \mathcal{P}(\mathcal{T}_\Sigma) \cup \{true, false\} \cup \mathcal{P}\left(\Sigma^0 \times \Sigma^0 \times \mathcal{T}_\Sigma\right)$ the set of *possible data values*, i.e., natural numbers, ground messages, sets of ground messages, Boolean values and sets of triples (agent, agent, message) for the *witness* predicate. □

Note that here, in order to simplify the presentation, we do not use an explicitly typed model. However, the implementation described in Section 5 does make use of a typed model in order to improve the efficiency of the tool (at the relatively small expense of not being able to find type-flaw attacks, which are anyway often "corrected" when moving from a protocol's specification to its concrete implementation in a typed programming language).

Definition 3. A *(SiL concrete) data state* (that we will sometimes refer to only as "state") is a function $\varsigma : V \to D$ and we denote with $\mathbb{D}$ the set of all such functions. □

In order to specify the behavior of SiL constructs, we present a *big-step structural operational semantics* for it. As it is the case for any structural operational semantics, the definition is given by means of a proof system. Rules manipulate judgments of the form $< T, \varsigma > \Downarrow v$, where T denotes an element in

[3] Two remarks are in order. First, for simplicity, we give the syntax in the case of a single goal to be considered; in case of more goals, a distinct *attack* variable can be added for each goal. Second, by the definition of the translation procedure into a SiL program, an authentication goal is verified immediately after the receipt of the message on which the authentication is based. Thus, we do not need in SiL an equivalent of the ASLan predicate *request*.

$$
\begin{array}{lcl}
E & ::= & X \mid c \\
M & ::= & E \mid [M,M] \mid \{M\}_M \mid \{M\}_{inv(M)} \mid \{|M|\}_M \\
L & ::= & M \mid L,M \\
S & ::= & \{L\} \mid IK \mid S+S \\
B & ::= & true \mid false \mid IK \vdash M \mid E = E \mid witness(E,E,M) \mid not(B) \mid B\ or\ B \mid B\ and\ B \\
C & ::= & X := E \mid IK := S \mid attack := B \mid witness(E,E,M) := true \mid C;C \mid if\ B\ then\ C\ else\ C \mid skip \mid end
\end{array}
$$

Fig. 4. The grammar of SiL.

any of the syntactic categories of SiL and v is a data value of the corresponding type (in particular, v is a state in the case when T is a statement). In a big-step semantics [29] formulation, $< T, \varsigma > \Downarrow v$ means that by the complete evaluation of T in the state ς, we obtain v. (This is in opposition to what happens in the case of the so-called small-step semantics, where each sequent denotes a minimal, atomic step of evaluation.) For instance, $< C, \varsigma > \Downarrow \varsigma'$ denotes that by evaluating the statement C in a state ς, we move to a state ς'. Given the simplicity of the language and the kind of analysis that we intend to carry out on it, we chose to give a big-step semantics, which typically has the advantage of needing fewer inference rules and allowing for a more concise presentation.

Definition 4. The *big-step operational semantics of SiL* is defined by the proof system in Fig. 5, where we use the following meta-variables: m ranges over $\mathcal{T}_\Sigma$, l ranges over lists of elements of $\mathcal{T}_\Sigma$, p ranges over $\mathcal{P}(\mathcal{T}_\Sigma)$, and $b \in \{true, false\}$. We denote with $\varsigma[m/X]$ the state obtained from ς by replacing the content of X by m, i.e., $\varsigma[m/X](Y) = m$ if $Y = X$ and $\varsigma[m/X](Y) = \varsigma(Y)$ otherwise. □

The rules for the evaluation of basic terms are quite simple: a constant evaluates to itself and a variable to the value associated to it in a given data state. The rules for compound messages evaluate the single components and then merge the results in a message of the appropriate form. The rules of the third class show how lists are evaluated by concatenating single messages and how sets of messages are built by using lists. In particular, the special set variable *IK* is evaluated in the same way as any other variable.

Evaluation of Booleans is standard: constants evaluate to themselves; predicates (equality and witness) evaluate either to *true* or *false*, according to a side condition referring to the values of the arguments; compound Boolean expressions are evaluated by functionally composing the truth values of the components.

Finally, the rules for statements modify the data state on which they are applied. Assignments modify the state value of the variable considered (be it a generic variable, *IK* or a variable referring to a predicate). Concatenation and conditional statements are treated as usual. *skip* and *end* do not alter the data state: the first one is just introduced in order to simplify the proof of some results, while the latter allows one to ignore the statements that follow.

3.2. The translation procedure

Definition 5. Given a protocol $\mathcal{P}$ involving a set $\mathcal{R}$ of *roles* ($Alice, Bob, \ldots$, a.k.a. *entities*), a *session instance* (or *session*, for short) *of* $\mathcal{P}$ is a function *si* assigning an agent (honest agent or the intruder *i*) to each element of $\mathcal{R}$. A *scenario of a protocol* $\mathcal{P}$ is a finite number of session instances of $\mathcal{P}$. □

The input of our method is then:

(1) an ASLan++ specification of a protocol $\mathcal{P}$,

BASIC TERMS

$$< X, \varsigma > \Downarrow \varsigma(X) \qquad < c, \varsigma > \Downarrow c$$

MESSAGES

$$\frac{< M_1, \varsigma > \Downarrow m_1 \quad < M_2, \varsigma > \Downarrow m_2}{< [M_1, M_2], \varsigma > \Downarrow [m_1, m_2]} \qquad \frac{< M_1, \varsigma > \Downarrow m_1 \quad < M_2, \varsigma > \Downarrow m_2}{< \{M_1\}_{M_2}, \varsigma > \Downarrow \{m_1\}_{m_2}}$$

$$\frac{< M_1, \varsigma > \Downarrow m_1 \quad < M_2, \varsigma > \Downarrow m_2}{< \{M_1\}_{inv(M_2)}, \varsigma > \Downarrow \{m_1\}_{inv(m_2)}} \qquad \frac{< M_1, \varsigma > \Downarrow m_1 \quad < M_2, \varsigma > \Downarrow m_2}{< \{|M_1|\}_{M_2}, \varsigma > \Downarrow \{|m_1|\}_{m_2}}$$

LISTS AND SETS OF MESSAGES

$$\frac{< L, \varsigma > \Downarrow l \quad < M, \varsigma > \Downarrow m}{< L, M, \varsigma > \Downarrow l, m} \qquad \frac{< L, \varsigma > \Downarrow l}{< \{L\}, \varsigma > \Downarrow \{l\}} \qquad < IK, \varsigma > \Downarrow \varsigma(IK) \qquad \frac{< S_1, \varsigma > \Downarrow p_1 \quad < S_2, \varsigma > \Downarrow p_2}{< S_1 + S_2, \varsigma > \Downarrow p_1 \cup p_2}$$

BOOLEAN EXPRESSIONS

$$\frac{< IK, \varsigma > \Downarrow \varsigma(IK) \quad < M, \varsigma > \Downarrow m}{< IK \vdash M, \varsigma > \Downarrow true} \; m \in DY(\varsigma(IK)) \qquad \frac{< IK, \varsigma > \Downarrow \varsigma(IK) \quad < M, \varsigma > \Downarrow m}{< IK \vdash M, \varsigma > \Downarrow false} \; m \notin DY(\varsigma(IK))$$

$$\frac{< E_1, \varsigma > \Downarrow c_1 \quad < E_2, \varsigma > \Downarrow c_2}{< E_1 = E_2, \varsigma > \Downarrow true} \; c_1 = c_2 \qquad \frac{< E_1, \varsigma > \Downarrow c_1 \quad < E_2, \varsigma > \Downarrow c_2}{< E_1 = E_2, \varsigma > \Downarrow false} \; c_1 \neq c_2$$

$$< true, \varsigma > \Downarrow true \qquad \frac{< E_1, \varsigma > \Downarrow c_1 \quad < E_2, \varsigma > \Downarrow c_2 \quad < M, \varsigma > \Downarrow m}{< witness(E_1, E_2, M), \varsigma > \Downarrow true} \; (c_1, c_2, m) \in \varsigma(witness)$$

$$< false, \varsigma > \Downarrow false \qquad \frac{< E_1, \varsigma > \Downarrow c_1 \quad < E_2, \varsigma > \Downarrow c_2 \quad < M, \varsigma > \Downarrow m}{< witness(E_1, E_2, M), \varsigma > \Downarrow false} \; (c_1, c_2, m) \notin \varsigma(witness)$$

$$\frac{< B, \varsigma > \Downarrow b}{< not(B), \varsigma > \Downarrow \neg b} \qquad \frac{< B_1, \varsigma > \Downarrow b_1 \quad < B_2, \varsigma > \Downarrow b_2}{< B_1 \; or \; B_2, \varsigma > \Downarrow b_1 \vee b_2} \qquad \frac{< B_1, \varsigma > \Downarrow b_1 \quad < B_2, \varsigma > \Downarrow b_2}{< B_1 \; and \; B_2, \varsigma > \Downarrow b_1 \wedge b_2}$$

STATEMENTS

$$\frac{< E, \varsigma > \Downarrow c}{< X := E, \varsigma > \Downarrow \varsigma[c/X]} \qquad \frac{< S, \varsigma > \Downarrow p}{< IK := S, \varsigma > \Downarrow \varsigma[p/IK]} \qquad \frac{< B, \varsigma > \Downarrow b}{< attack := B, \varsigma > \Downarrow \varsigma[b/attack]}$$

$$\frac{< E_1, \varsigma > \Downarrow c_1 \quad < E_2, \varsigma > \Downarrow c_2 \quad < M, \varsigma > \Downarrow m}{< witness(E_1, E_2, M) := true, \varsigma > \Downarrow \varsigma[witness \cup \{(c_1, c_2, m)\}/witness]} \qquad \frac{< C_0, \varsigma > \Downarrow \varsigma'' \quad < C_1, \varsigma'' > \Downarrow \varsigma'}{< C_0; C_1, \varsigma > \Downarrow \varsigma'}$$

$$\frac{< B, \varsigma > \Downarrow true \quad < C_0, \varsigma > \Downarrow \varsigma'}{< if \; B \; then \; C_0 \; else \; C_1, \varsigma > \Downarrow \varsigma'} \qquad \frac{< B, \varsigma > \Downarrow false \quad < C_1, \varsigma > \Downarrow \varsigma'}{< if \; B \; then \; C_0 \; else \; C_1, \varsigma > \Downarrow \varsigma'}$$

$$< skip, \varsigma > \Downarrow \varsigma \qquad < end, \varsigma > \Downarrow \varsigma \qquad < end; C_1, \varsigma > \Downarrow \varsigma$$

Fig. 5. A big-step semantics for SiL.

(2) a scenario $\mathcal{S}$ of $\mathcal{P}$, and
(3) a set of goals (i.e., properties to be verified) in $\mathcal{S}$.

We will first describe how to obtain a program for a single session and then how to decorate it with goal locations used to verify security properties. In Section 3.3, finally, we will explain how to combine more sessions in a single program.

3.2.1. Translating a single session

First of all, we notice that in our translation, and according to the ASLan++/ASLan instantiation mechanism, a session instance between two honest agents is represented as the composition of two sessions, where each of the honest agents communicates with the intruder. We will refer to the session instances obtained after such a translation as *program instances*.

Example 4. *For example, the second session of our running example (Example 1), i.e., the one between a and b, is obtained by the composition of two program instances, the first played by a and $i(b)$ and the second by $i(a)$ and b, thus giving rise to the following three program instances*

Program	*Alice*	*Bob*
1	*a*	*i*
2	*a*	*i(b)*
3	*i(a)*	*b*

□

To simplify notation, for the variables and constants of the resulting program we will use the same names as the ones used in the ASLan++ specification. However, in order to distinguish between variables with the same name occurring in the specification of different roles, program variables have the form *E.V*, where *E* denotes the role and *V* the variable name in the specification. In the case when more than one session are considered, we also prefix an index denoting the session to the program variable name, e.g., as in *S1_E.V*.

The behavior of the intruder introduces a form of non-determinism even within a single session, e.g., related to the construction of a message sent by the intruder, which we capture by letting the program depend on a number of input values, one for each intruder choice. The corresponding input variables are denoted by the symbol *Y*, possibly subscripted with an index. Finally, symbols of the form *c_i*, for *i* an integer, are used to denote constants to be assigned to nonces.

Structure of the program. The exchange of messages in a session follows a given flow of execution that can be used to determine an order between the instructions contained in the different roles. Such a sequence of instructions will constitute the skeleton of the program.

After a first section that concerns the initialization of the variables, the program will indeed contain a proper translation, based on the semantics of ASLan++, of the instructions in such a sequence. For each program instance, we will follow the flow of execution of the honest agents, as we can think of the intruder actions as not being driven by any protocol, and model the intruder interaction with the honest agents by means of $IK \vdash M$ statements and updates of IK.

In the next paragraphs, we will describe more specifically: (i) how variables are initialized and (ii) how each statement is translated.

Initialization of the variables. A first section of the program consists of the initialization of the variables. Let *pi* be the program instance of the program we are considering. For each role *Alice* such that $pi(Alice) = a$, for some agent name $a \neq i$, we have an initialization instruction $Alice.Actor := a$. Furthermore, for the same *Alice*, and for each other role *Bob*, with *B* being the variable referring to the role *Bob* amongst the agent variables of *Alice*: if $si(Bob) = b$, then we have the assignment $Alice.B := b$. Finally, it is necessary to initialize the intruder knowledge. A typical initialization instruction for *IK* has the form:

$$IK := \{a\_1, ..., a\_n, i, pk(a\_1), ..., pk(a\_n), pk(i), inv(pk(i))\}.$$

That is, *i* knows each agent *a_j* involved in the scenario and his public keys $pk(a\_j)$, as well as his own public and private keys $pk(i)$ and $inv(pk(i))$. Specific protocols might require a specific initial intruder knowledge or the initialization of further variables, depending on the context, such as symmetric keys or hash functions, which are possibly defined in the Prelude section of the ASLan++ specification.

Sending of a message. The sending of a message $Actor \rightarrow B : M$ defined in a role *Alice* is translated into the instruction $IK := IK + \{M\}$, where the symbol $+$ denotes set union (corresponding to $\cup$) so that the the message *M* is added to the intruder knowledge.

Receipt of a message. Consider the receipt of a message $R \rightarrow Actor : M$ in a role *Alice*. Assume the message is sent from a role *Bob*. Then the instruction is translated into the following code

```
1 If (IK |- M)
2   then Alice.Q_1 := Y_1;
3        ...
4        Alice.Q_n := Y_n;
5   else end;
```

where $Q\_1, ..., Q\_n$ are all the variables preceded by ? occurring in *M* and $Y\_1, ..., Y\_n$ are distinct input variables not introduced elsewhere.

Generation of fresh values. Finally, an instruction of the form $N := fresh()$ in *Alice*, which assigns a fresh value to a nonce, can be translated into the instruction $Alice.N := c\_1$, where *c_1* is a constant not introduced elsewhere.

3.2.2. Defining goals for the verification of security properties

Introducing goal locations. The next step consists of decorating the program with a goal location for each security property to be verified. As it is common when performing symbolic execution [20], we express such properties as correctness assertions, typically placed at the end of a program. Once we have represented a protocol session as a program (or more programs in the case when a session instance is split into more program instances), and defined the properties we are interested in as correctness assertions in such a program, the problem of verifying security properties over (a session of) the protocol is reduced to verifying the correctness of the program with respect to those assertions.

We consider here two common security properties (authentication and confidentiality) and show how to represent them in the program in terms of assertions. They are expressed by means of a statement of the form $if\ (not(expr))\ then\ attack := true$, where *expr* is an expression referring to the goal considered, as described below.

Authentication. Assume that we want to verify that *Alice* authenticates *Bob* with respect to a message M in the specification of the protocol, in a given program instance by the ASLan++ statement: $B \rightarrow Actor : auth : (M)$, where *auth* is the label of the goal and a corresponding sending statement is included in the specification.

We can restrict our attention to the case when according to the program instance under consideration *Bob* is played by i, since otherwise the authentication property is trivially satisfied. The problem thus reduces to verifying whether the agent i is playing under his real name (in which case authentication is again trivially satisfied) or whether i is pretending to be someone else, i.e., whether the agent playing *Alice* believes she is speaking to someone who is not i. Hence, one of the conditions required in order to reach the goal is $not(Alice.B = i)$, where B is the agent variable referring to the role *Bob* inside *Alice*.

A second condition is necessary and concerns the fact that the message M has not been sent by $Alice.B$ to $Alice.Actor$. This can be verified by using the witness predicate, which is set to true when the message is sent and whose state is checked when a goal is searched for, i.e., immediately after the receipt of the message M.

Example 5. *In NSL, we are interested in verifying a property of authentication in the session that assigns i to Alice and b to Bob: namely, we want Bob to authenticate Alice with respect to the nonce Bob.Nb contained in the reception in line 8 on the right of the NSL example (Example 1). Such a receipt corresponds to the sending of line 8 on the left. Thus we can add a witness assignment of the form |witness(Alice.Actor, Alice.B, [Alice.Nb,pk(Alice.B)]) := true| after the sending, and the instruction*

```
1 if (not(Bob.A = i) and not(witness(Bob.A, Bob.Actor, {Bob.Nb}_pk(Bob.Actor))))
2   then attack:=true;
3   else skip;
```

after the receipt of the message. □

Confidentiality. Assume that we want to verify that the message corresponding to a variable M, in the specification of a role *Alice* of the protocol, is confidential between a given set of roles $\mathcal{R} = \{Alice\_1, \ldots, Alice\_n\}$ in a session si, i.e., we have a sending statement $Actor \rightarrow B : \{secret : (M)\}$, where *secret* is the goal label, for a confidentiality goal expressed as $secret : (\_)\ \{Alice\_1, \ldots, Alice\_n\}$. This amounts to checking whether the agent i got to know the confidential message M even though i is not included in $\mathcal{R}$. Inside the program, this corresponds to verifying whether the message $Alice.M$ can be derived from the intruder knowledge and whether any honest agent playing a role in $\mathcal{R}$ believes that at least one of the other roles in $\mathcal{R}$ is indeed played by i, which we can read as having indeed $i \in \mathcal{R}$. The following assertion is added at the end of the SiL program:

```
1 if ((IK |- Alice.M) and (not((Alice_1.B^1_1 = i) or
2       ... (Alice_1.B^1_m = i) or ...
3       (Alice_n.B^n_1 = i) or ... (Alice_n.B^n_m = i))))
4   then attack := true;
5   else skip;
```

where $Alice\_j$, for $1 \leqslant j \leqslant n$, is a role such that $Alice\_j \in \mathcal{R}$ and $si(Alice\_j) \neq i$, $\{Bob\_1, \ldots, Bob\_m\} \subseteq \mathcal{R}$ is the subset of those roles in $\mathcal{R}$ that are instantiated with i by si and B^j_l, for $1 \leqslant j \leqslant n$ and $1 \leqslant l \leqslant m$, is the variable referring to the role Bob_l in the specification of the role $Alice\_j$.

Example 6. *For NSL, assume that we want to verify the confidentiality of the variable Nb (contained in the specification of Bob) between the roles in the set* $\{Alice, Bob\}$*. We can express this goal by appending the assertion*

```
if ((IK |- Bob.Nb) and (not(Bob.A = i)))
  then attack := true;
  else skip;
```

at the end of the program. □

Example 7. *The program instances described in Example 4 give rise to the following three SiL programs, which have a single IK initialization instruction:*

```
IK := {a,b,i,pk(a),pk(b),pk(i),inv(pk(i))}
```

Program 1

```
S1_Alice.Actor := a;
S1_Alice.B := i;
S1_Alice.Na := c_0;

IK := IK + {{S1_Alice.Na,S1_Alice.Actor}_pk(S1_Alice.B)};

if (IK |- {S1_Alice.Na,[S1_Alice.Y_1,S1_Alice.B]}_pk(S1_Alice.Actor)
  then S1_Alice.Nb := S1_Alice.Y_1;
  else end;

IK := IK + {{S1_Alice.Nb}_pk(S1_Alice.B)};
witness(S1_Alice.Actor,S1_Alice.B,{S1_Alice.Nb}_pk(S1_Alice.B)) := true;
```

Program 2

```
S2_Alice.Actor := a;
S2_Alice.B := b;

S2_Alice.Na := c_1;

IK := IK + {{S2_Alice.Na,S2_Alice.Actor}_pk(S2_Alice.B)};

if (IK |- {S2_Alice.Na,[S2_Alice.Y_1,S2_Alice.B]}_pk(S2_Alice.Actor))
  then S2_Alice.Nb := S2_Alice.Y_1;
  else end;

IK := IK + {{S2_Alice.Nb}_pk(S2_Alice.B)};
witness(S2_Alice.Actor,S2_Alice.B,{S2_Alice.Nb}_pk(S2_Alice.B)) := true;
```

Program 3

```
S2_Bob.A := a;
S2_Bob.Actor := b;

if (IK |- {S2_Bob.Y_1,S2_Bob.Y_2}_pk(S2_Bob.Actor))
  then S2_Bob.Na := S2_Bob.Y_1;
       S2_Bob.A := S2_Bob.Y_2;
  else end;

S2_Bob.Nb := c_2;

```

```
11  IK := IK + {{S2_Bob.Na,[S2_Bob.Nb,S2_Bob.Actor]}_pk(S2_Bob.A)};
12
13  if (IK |- {S2_Bob.Nb}_pk(S2_Bob.Actor))
14    then
15      if  (not(witness(S2_Bob.A, S2_Bob.Actor, {S2_Bob.Nb}_pk(S2_Bob.Actor)))
16          and
17          (not(S2_Bob.A = i)));
18        then attack := true;
19    else end;
```

□

3.3. Combining sessions

Now we need to define a global program that properly "combines" the programs related to all the sessions in the scenario. The idea is that such a program allows for executing, in the proper order, all the instructions of all the sessions in the scenario; the way in which instructions of different sessions are interleaved will be determined by the value of further input variables, denoted by X (possibly subscripted), which can be seen as choices of the intruder with respect to the flow of the execution. Namely, we start to execute each session sequentially and we get blocked when we encounter the receipt of a message sent by a role that is played by the intruder. When all the sessions are blocked on instructions of that form, the intruder chooses which session has to be reactivated (by setting the variables X accordingly).

For what follows, it is convenient to see a sequential program as a graph (which can be simply obtained by representing its control flow) on which the algorithm of Section 4 for symbolic execution and annotation will be executed. We recall here some notions concerning programs and program runs.

Definition 6. A *(SiL) program graph* is a finite, rooted, labeled graph (Λ, l_0, Δ), where Λ is a *finite set of program locations*, l_0 is the *initial location* and $\Delta \subseteq \Lambda \times \mathcal{A} \times \Lambda$ is a *set of transitions* labeled by actions from a set $\mathcal{A}$, containing the assignments and conditional statements provided by the language SiL.

A *(SiL) program path* of length k is a sequence of the form $l_0, a_0, l_1, a_1, \ldots, l_k$, where each *step* $(l_j, a_j, l_{j+1}) \in \Delta$ for $0 \leqslant j < k-1$.

Let ς_0 be the initial data state. A *(SiL) program run* of length k is a pair (π, ω), where π is a program path $l_0, a_0, l_1, a_1, \ldots, l_k$ and $\omega = \varsigma_0, \ldots, \varsigma_{k+1}$ is a sequence of data states such that $< a_j, \varsigma_j > \Downarrow \varsigma_{j+1}$ for $0 \leqslant j \leqslant k$. □

Let $\mathcal{S}$ be a scenario of a protocol $\mathcal{P}$ with m program instances $pi_1, \ldots, pi_m$. We can associate to each program instance pi_j, for $1 \leqslant j \leqslant m$, a sequential program by following the procedure described in Section 3.2.

For each $1 \leqslant j \leqslant m$, we have a program graph $\mathcal{G}^j = (\Lambda^j, l_0^j, \Delta^j)$ corresponding to the program of pi_j. The program graph $\mathcal{G}$ corresponding to a given scenario is obtained by composing the graphs of the single program instances. Below we describe an algorithm for concretely obtaining such a program graph for $\mathcal{S}$. For simplicity, we will assume that the original specification of $\mathcal{P}$ is such that no receipts of messages are contained inside an if-statement.

Definition 7. Given a program graph, an *intruder location* is a location of the program graph corresponding to the receipt of a message.

A *block of a program graph* $\mathcal{G}'$ is a subgraph of $\mathcal{G}'$ such that its initial location is either the initial location of $\mathcal{G}'$ or an intruder location.

The *exit locations of a block* $\mathcal{B}$ are the locations of $\mathcal{B}$ with no outgoing edges.

A program graph can simply be seen as a sequence of blocks. Namely, we can associate to the program graph $\mathcal{G}^j$, for each $1 \leqslant j \leqslant m$, its *block structure*, i.e., a sequence $\mathcal{B}^j_1, \ldots, \mathcal{B}^j_n$ of blocks of $\mathcal{G}^j$, such that: (i) the initial location of $\mathcal{B}^j_1$ is the initial location of $\mathcal{G}^j$; (ii) each intruder location of $\mathcal{G}^j$ is the initial location of $\mathcal{B}^j_k$ for some $1 \leqslant k \leqslant m$; (iii) for $1 \leqslant k < n$, the initial location of $\mathcal{B}^j_{k+1}$ coincides, in $\mathcal{G}^j$, with an exit location of $\mathcal{B}^j_k$; (iv) the program graph obtained by composing $\mathcal{B}^j_1, \ldots, \mathcal{B}^j_n$, i.e., by letting the initial location of $\mathcal{B}^j_{k+1}$ coincide with the corresponding exit location of $\mathcal{B}^j_k$, is $\mathcal{G}^j$ itself. □

Intuitively, we decompose a session program graph $\mathcal{G}^i$ into sequential blocks starting at each intruder location. For instance, Program 1 of Example 7 can be divided into two parts giving rise to two distinct blocks:

Block $\mathcal{B}^1_1$

```
1  S1_Alice.Actor := a;
2  S1_Alice.B := i;
3  S1_Alice.Na := c_0;
4  IK := IK + {{S1_Alice.Na,S1_Alice.Actor}_pk(S1_Alice.B)};
```

Block $\mathcal{B}^1_2$

```
1  if (IK |- {S1_Alice.Na,[S1_Alice.Y_1,S1_Alice.B]}_pk(S1_Alice.Actor)
2    then S1_Alice.Nb := S1_Alice.Y_1;
3    else end;
4
5  IK := IK + {{S1_Alice.Nb}_pk(S1_Alice.B)};
6  witness(S1_Alice.Actor,S1_Alice.B,{S1_Alice.Nb}_pk(S1_Alice.B)}) := true;
```

The idea is that each such block will occur as a subgraph in the general scenario program graph $\mathcal{G}$ (possibly with more than one occurrence). Namely, the procedure for generating the scenario program graph will create a program graph that allows one to execute all the blocks of the scenario just once, in any possible sequence that respects the order of the single sessions, i.e., each possible interleaving of blocks will be considered. For instance, if we assume to have the block structures $(\mathcal{B}^1_1, \mathcal{B}^1_2)$ and $(\mathcal{B}^2_1)$, the resulting program graph will contain a path corresponding to the execution of $\mathcal{B}^1_1, \mathcal{B}^1_2, \mathcal{B}^2_1$ in this order, as well as a path for $\mathcal{B}^1_1, \mathcal{B}^2_1, \mathcal{B}^1_2$, as well as a path for $\mathcal{B}^2_1, \mathcal{B}^1_1, \mathcal{B}^1_2$. Given a block, its *main exit location* is a location with no outgoing edges such that in the original session graph it has an outgoing edge towards an intruder location. Note that under the restriction on $\mathcal{P}$ introduced above (i.e., no receipts inside if-statements), each block has at most one such location, while other (non-main) exit locations may arise for the presence of *end* statements.

In Fig. 6, we give an algorithm that we have devised to incrementally build the program graph $\mathcal{G} = (\Lambda, l_0, \Delta)$ starting from the root and adding blocks step by step. We assume the number of program instances m given. In the algorithm we use a procedure *attach*, which given a block $\mathcal{B}$ and a location l, adds the subgraph $\mathcal{B}$ to $\mathcal{G}$ (by letting the initial location of $\mathcal{B}$ coincide with l) and updates the sets Λ and Δ accordingly. During the construction, the set $T \subseteq \Lambda$ contains the locations of the program graph to be still expanded. Two functions $pc : \Lambda \times \{1, \ldots, m\} \to \mathbb{N}$ and $ic : \Lambda \to \mathbb{N}$ are used to keep track of the status of the construction. Their intended meaning is the following: assume that the location l in the program graph is still to be expanded; then for each $1 \leqslant j \leqslant m$, $\mathcal{B}^j_{pc(l,j)}$ is the next block to be added for what concerns the program instance pi_j (i.e., each path going from the root to l has already executed $\mathcal{B}^j_h$ for $1 \leqslant h < pc(l, j)$) and the next input variable to be used is $X_{ic(l)}$.

Input: The set of session program graphs $\{\mathcal{G}^1,\ldots,\mathcal{G}^m\}$ related to a given scenario of a protocol. For $1 \leqslant j \leqslant m$, we refer to the *block structure* of $\mathcal{G}^j$ as the sequence $\mathcal{B}^j_1,\ldots,\mathcal{B}^j_{n_j}$.
Output: The program graph $\mathcal{G} = (\Lambda, l_0, \Delta)$ combining $\mathcal{G}^1,\ldots,\mathcal{G}^m$.

create a location l;
$\Lambda := \{l\}$;
$l_0 := l$;
$\Delta := \emptyset$;
$pc(l,j) := 1$ for $1 \leqslant j \leqslant m$;
$ic(l) := 1$;
for $h = 1$ to m do {
  if (initial location of $\mathcal{B}^h_1$ is not intruder location)
  then {
    attach $\mathcal{B}^h_1$ to l;
    let l' be the main exit location of $\mathcal{B}^h_1$:
    $pc(l',j) := pc(l,j)$ for all $j \neq h$;
    $pc(l',h) := pc(l,h)+1$;
    $ic(l') := 1$;
    $l := l'$
  }
}
$T := \{l\}$;
do {
  pick a location $l \in T$;
  for $h = 1$ to m do {
    if ($\mathcal{B}^h_{pc(l,h)}$ does exist) then {
      create a location l^*;
      $\Lambda := \Lambda \cup \{l^*\}$;
      $\Delta := \Delta \cup \{(l,$ X_k = h$,l^*)\}$, where $k = ic(l)$;
      attach $\mathcal{B}^h_{pc(l,h)}$ to l^*;
      let l' be the main exit location of $\mathcal{B}^h_{pc(l,h)}$:
      $pc(l',j) := pc(l,j)$ for all $j \neq h$;
      $pc(l',h) := pc(l,h)+1$;
      $ic(l') := ic(l)+1$;
      $T := T \cup \{l'\}$;
    }
  }
  $T := T \setminus \{l\}$;
} while ($T \neq \emptyset$);

Fig. 6. An algorithm for building the program graph combining more sessions.

The first **for** loop in the pseudo-code of the algorithm composes, in a sequence, the first blocks of each session program graph. Then the **while** loop expands the program graph by adding a fork at each intruder choice.

The resulting program graph $\mathcal{G} = (\Lambda, l_0, \Delta)$, which is actually a tree, can be finally simplified by collapsing indistinguishable nodes, according to standard graph and transition systems optimization techniques based on minimization modulo bisimulation, as well as by omitting paths that do not lead to any goal location.

Example 8. *Fig. 10 shows the message sequence chart corresponding to one of the paths of the program graph for NSL, in the scenario described in the previous examples. The entire graph (whose block composition is shown in Fig. 7) is obtained by using the algorithm of Fig. 6 plus some optimization, as described in the text above. The path highlighted in double lines in Fig. 7 is the one shown in Fig. 10.* □

3.4. Correctness of the translation

Now, we show that the translation into SiL, defined in Sections 3.2 and 3.3, preserves important properties of the original specification. In particular, we show that given an ASLan++ specification, an attack state can be reached by analyzing its ASLan translation if and only if an attack state can be found by executing its SiL translation.

Equivalence of single steps.

Definition 8. We say that an ASLan term M' and a SiL term M'' are *equivalent*, $M' \sim M''$, iff one of the following conditions holds:

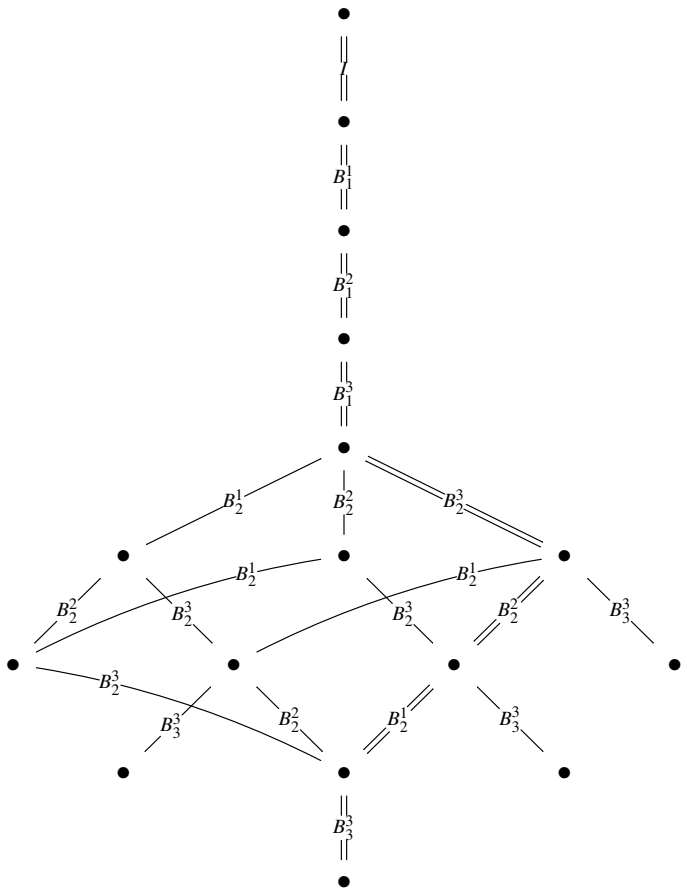


Fig. 7. A SiL program graph for NSL

- $M' \equiv c'$, $M'' \equiv c''$ and $c' = c''$;
- $M' \equiv pair(M'_1, M'_2)$, $M'' \equiv [M''_1, M''_2]$ and $M'_1 \sim M''_1$, $M'_2 \sim M''_2$;
- $M' \equiv crypt(M'_1, M'_2)$, $M'' \equiv \{M''_2\}_{M''_1}$ and $M'_1 \sim M''_1$, $M'_2 \sim M''_2$;
- $M' \equiv scrypt(M'_1, M'_2)$, $M'' \equiv \{|M''_2|\}_{M''_1}$ and $M'_1 \sim M''_1$, $M'_2 \sim M''_2$;
- $M' \equiv inv(M'_1)$, $M'' \equiv inv(M''_1)$ and $M'_1 \sim M''_1$.

where $\equiv$ denotes syntactic equality. □

In the following, we consider an ASLan++ program and the corresponding ASLan translation. In order to do that, we will define and use some auxiliary functions that will help relate ASLan and SiL notions. First of all, as described in Section 2.3, we recall that for each predicate symbol in the *SignatureSection* we will have a corresponding state fact.

Definition 9. We define a *variable mapping* as a function $f(E,A)$ that given an entity name E and a variable name A returns the value i corresponding to the index of the position of variable A in the state fact $state\_E$. □

Note that such a function always exists and it is implicitly created at translation time by the translation procedure from ASLan++ into ASLan described in Section 2.3.

Let $pi_1, \ldots, pi_n$ be the program instances of the considered protocol scenario and let S be any ASLan state in the corresponding ASLan description. We can assume to have a further function g that will be used to denote the identifier of a given session instance. Namely, we define $g(j) = SID$, where SID is the identifier contained in the state fact $state\_Session\_j(\ldots, SID, \ldots) \in S$, i.e., the state fact that represents in S the symbolic session corresponding to the program instance pi_j. Note that such a function is implicitly created when a symbolic session is instantiated (Section 2.3) and it is bijective. Furthermore, we introduce some notation in order to be able to refer to specific values in the state of an arbitrary entity E. Namely, given a session instance j, we will write $S(E_j, i)$ to denote the value v_i of the state predicate $state\_E(v_1, ID, \ldots, v_n)$ such that $child(g(j), ID) \in S$. The last condition on the predicate *child* ensures that we refer to the value of the entity in the appropriate session instance (j in this case).

Definition 10. We say that an ASLan state S and a SiL state ς are *equivalent*, $S \sim \varsigma$, iff:

- for each SiL ground term M' and ASLan ground term M'' such that $M' \sim M''$, $M' \in DY(\varsigma(IK)) \Leftrightarrow iknows(M'') \subseteq \lceil S \rceil^H$;
- $\varsigma(Sj\_E.A) = S(E_j, f(E,A))$ for each E representing an entity name involved in the protocol, for each A representing an ASLan++ variable name or parameter name of entity E, for each session instance si_j
- $\varsigma(attack) = true \Leftrightarrow attack \subseteq \lceil S \rceil^H$;
- $(M, M_1, M_2) \in \varsigma(witness) \Leftrightarrow witness(M', M'_1, M'_2, \ldots) \subseteq \lceil S \rceil^H$, where M, M_1 and M_2 are SiL ground terms and M', M'_1 and M'_2 are ASLan ground terms such that $M \sim M'$, $M_1 \sim M'_1$ and $M_2 \sim M'_2$. □

We notice that while an ASLan transition occurs when there exists a substitution (of values for variables) that makes a rule applicable, in SiL we simulate, and in a sense make more explicit, such a substitution by using the Y input variables. This establishes a correspondence between ASLan substitutions and assignments of values to SiL input variables, which will be important in the following proofs, and that we will handle by means of the following notion of *extension* of a SiL state.

Definition 11. Given a SiL state ς and a set of input variables $Y_1, \ldots, Y_n$ such that $\varsigma(Y_i)$ is undefined, we define an *extension* $\bar{\varsigma}$ *of* ς as a SiL state, where $\bar{\varsigma}$ is defined for $Y_1, \ldots, Y_n$ and for each other variable A, $\bar{\varsigma}(A) = \varsigma(A)$. □

Since the input variables of the form Y_i are not involved in the definition of equivalence, if an ASLan state S and a SiL state ς are equivalent (i.e., $S \sim \varsigma$), and $\bar{\varsigma}$ is an extension of ς, then also S and $\bar{\varsigma}$ are equivalent (i.e., $S \sim \bar{\varsigma}$).

Let r be an ASLan rule; we will write $S \xrightarrow{r} S'$ iff there exists a transition from an ASLan state S to an ASLan state S' obtained by applying the rule r.

Lemma 1. *Let I be an ASLan++ statement, r the corresponding ASLan rule and w the corresponding SiL code, as defined in Section 2.3 and 3.2, respectively. Given an ASLan state S and a SiL state* ς *such that* $S \sim \varsigma$ *we have:*

(1) If $S \xrightarrow{r} S'$ *then there exists an extension* $\bar{\varsigma}$ *of* ς *such that* $< w, \bar{\varsigma} > \Downarrow \varsigma'$ *and* $S' \sim \varsigma'$*;*

(2) If there exists an extension $\bar{\varsigma}$ of ς such that $< w, \bar{\varsigma} > \Downarrow \varsigma'$, then either there exists an S' such that $S \xrightarrow{r} S'$ and $S' \sim \varsigma'$ or $S \sim \varsigma'$.

Proof. The proof proceeds by considering all the possible ASLan++ statements and is given in Appendix B. □

Equivalence of runs. We have showed that, starting from equivalent states, the application of ASLan rules and SiL code fragments that have been generated from the same ASLan++ statements leads to states that are still equivalent. Now we will show that given an ASLan++ specification, for each run in the SiL translation, there exists a sequence of corresponding ASLan rules in the ASLan translation.

In order to compare SiL actions and ASLan rules, a few things need to be taken into account. The goal here is to define things in such a way that a step of execution in SiL corresponds exactly to a step of execution in ASLan. First of all, we note that, strictly speaking, the translation of an ASLan++ statement into SiL is not always an atomic action, e.g., in the case of a receipt, the corresponding SiL action comprises both a conditional and some assignments. This is not reflected in ASLan. In order to make an easier comparison with the corresponding ASLan step, we thus collect such blocks of actions into a single compound action. Moreover, if we consider a path in a SiL program graph, we encounter conditional statements referring to X_i variables, i.e., those used in SiL to handle the interleaving between sessions. These do not have a direct correspondent in terms of ASLan rules and will therefore not be included in the following definition of a SiL action path.

Definition 12. Assume given a program graph $\mathcal{G}$ for a protocol $\mathcal{P}$ and a scenario $\mathcal{S}$. A *(SiL) compound action* is a sequence of SiL actions that correspond altogether to the translation of a single ASLan++ statement. A *SiL action path (for $\mathcal{G}$)* is a sequence $w_0, \ldots, w_k$ of SiL compound actions that label, in the given order, the edges of a path of $\mathcal{G}$.

We define a *SiL action run (for $\mathcal{G}$)* as a pair (π, ω), where $\pi = w_0, \ldots, w_k$ is a SiL action path and $\omega = \varsigma_0, \ldots, \varsigma_{k+1}$ is a sequence of data states such that $< w_j, \varsigma_j > \Downarrow \varsigma_{j+1}$ for $0 \leqslant j \leqslant k$. □

We notice that the definition above does exclude X_i conditional statements as they do not come from the translation of an ASLan++ rule and thus they are not considered compound actions. Now it is easy to see that the notions of SiL program path and SiL action path are strictly related, as they both refer to a path obtained by interleaving the program chunks of different sessions. Intuitively, given a program graph, we have that to each SiL program path corresponds a SiL action path (obtained by "reading" the actions on the edges of the SiL program path, removing the X_i-conditionals and possibly grouping some consecutive atomic actions). The notion of action path is introduced because it allows for an easier comparison with paths obtained as sequences of ASLan rules, as defined in the following.

Definition 13. Assume given a protocol $\mathcal{P}$ and let $E_1, \ldots, E_n$ be the entity names involved in $\mathcal{P}$. We denote with $I_e \equiv I_{e,1}, \ldots, I_{e,m_e}$ the sequence of ASLan++ statements corresponding to the entity E_e.

Given a scenario $\mathcal{S}$, for each program instance $pi(j)$, we denote with $r^j_{e,1}, \ldots, r^j_{e,m_e}$ the sequence of ASLan rules corresponding to I_e.

An *ASLan path (for a protocol scenario $\mathcal{S}$)* is a sequence $r_0, \ldots, r_k$ of ASLan rules such that:

- for each entity E_e, program instance $pi(j)$ and $1 \leqslant l \leqslant m_e$, there is one and only one $0 \leqslant i \leqslant k$ such that $r_i \equiv r^j_{e,l}$;
- for $0 \leqslant i \leqslant k$, $r_i \equiv r^j_{e,l}$ for some e, l and j;

- for $0 \leqslant i \leqslant k$, if $state\_E(...,sl,...)$, where sl is the step label, is in the left-hand side of $r_i \equiv r^j_{e,l}$ then either $sl = 1$ or there exists $h < i$ such that $state\_E(...,sl,...)$ is in the right-hand side of r_h and $r_h \equiv r^j_{e,l-1}$. □

The intuition behind this definition is that, given an ASLan transition system, the set of ASLan paths collects all the "potential" sequences of applications of ASLan rules, i.e., those admissible by only taking care of respecting the order given by the step labels inside the rules, no matter how the rest of the state evolves. The condition on the step labels is used to ensure that rules belonging to a same session are applied in the correct order.

Definition 14. An *ASLan run (for a protocol scenario $\mathcal{S}$)* is a pair (τ,ρ), where τ is an ASLan path $r_0,\ldots,r_k$ and $\rho = S_0,\ldots,S_{k+1}$ is a sequence of ASLan states such that $S_i \xrightarrow{r_i} S_{i+1}$ for $0 \leqslant i \leqslant k$. □

Definition 15. We say that an ASLan path $r_0,\ldots,r_k$ and a SiL action path $w_0,\ldots,w_k$ are *equivalent* iff for each $0 \leqslant i \leqslant k$, r_i and w_i can be obtained as the translation of the same ASLan++ statement. □

Lemma 2. *Let $\mathcal{S}$ be a protocol scenario and $\mathcal{G}$ the corresponding program graph. Then: (i) for each SiL action path $w_0,\ldots,w_k$ for $\mathcal{G}$, there exists an equivalent ASLan path $r_0,\ldots,r_k$ for $\mathcal{S}$; and, conversely, (ii) for each ASLan path $r_0,\ldots,r_k$ for $\mathcal{S}$, there exists an equivalent SiL action path $w_0,\ldots,w_k$ for $\mathcal{G}$.*

Proof. It is enough to observe that SiL action paths and ASLan paths follow, for a given program instance, the order in which the actions are executed in the protocol: this is obtained by the definition of the graph construction in the case of SiL, and by using step labels inside the rules in the case of ASLan. Furthermore, in both cases, each possible interleaving between sessions is admitted, i.e., whenever in a SiL path an action of the program instance $pi(i)$ is followed by an action of the program instance $pi(j)$, there is a corresponding possible choice for a next rule r to be applied in ASLan such that $r = r^j_{e,l}$ for some e and l; conversely, for each ASLan rule in an ASLan path letting one switch from a session i to a session j, there is a corresponding branch where $X_h = j$ giving rise to a corresponding SiL path. □

Theorem 1. *For each SiL action run (π,ω) of graph $\mathcal{G}$ corresponding to the protocol scenario $\mathcal{S}$, where $\omega = \varsigma_0,\ldots,\varsigma_{k+1}$, there exists an ASLan run (τ,ρ) for $\mathcal{S}$, where $\rho = S_0,\ldots,S_{k+1}$, and $\varsigma_i \sim S_i$ for $0 \leqslant i \leqslant k+1$. The converse also holds.*

Proof. Let ς_0 be the data state obtained after the initialization block of the SiL program graph and S_0 the ASLan initial state, as defined in Section 2. It is easy to check that $\varsigma_0 \sim S_0$. Then, the thesis follows by using Lemma 2 (for each SiL action path, there is an equivalent ASLan path, and vice versa) and Lemma 1 (equivalent steps preserve equivalence of states). □

Finally, we can use the previous theorem to show that an attack state can be found in an ASLan path iff a goal location can be reached in the corresponding SiL path.

Corollary 1. *Let $\mathcal{S}$ be a protocol scenario and $\mathcal{G}$ the corresponding program graph. An attack state can be found in an ASLan path for $\mathcal{S}$ iff a goal location can be reached in a SiL action path for $\mathcal{G}$.*

Proof. Let S be an ASLan attack state, i.e., $attack \subseteq \lceil S \rceil^H$. By Theorem 1, S is in an ASLan run for $\mathcal{S}$ iff there exists $\varsigma \sim S$ in a SiL action run for $\mathcal{G}$. By Definition 10, $\varsigma(attack) = true$, i.e., a goal location

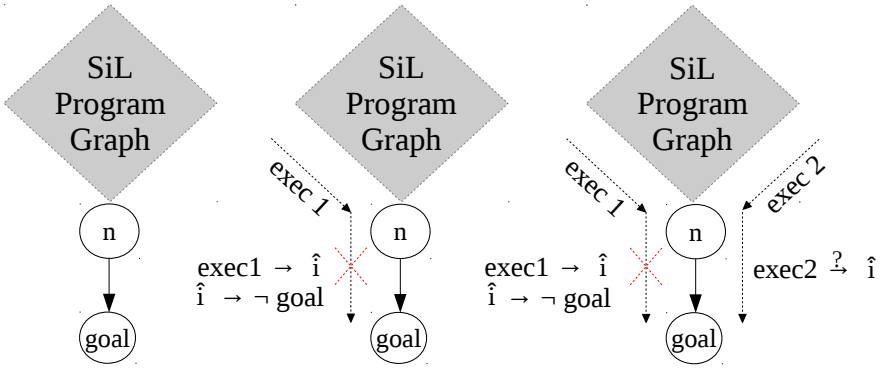


Fig. 8. A SiL program graph (left). A first phase of symbolic execution with generation of an annotation (center). A second phase of symbolic execution with annotation check (right).

referring to the given attack has been reached. Since Theorem 1 holds in both directions, the converse is also proved. □

4. An interpolation-based algorithm for verification

In this section, we present the interpolation-based algorithm that we use for verification and describe, in particular, how we can calculate interpolants in our specific setting.

Our algorithm is a slightly simplified version of the IntraLA algorithm of [15], obtained by removing some fields only used there to deal with program procedures. In a nutshell, the idea underlying our algorithm is as follows. The input of our algorithm is a SiL program graph, as defined in Section 3.3, together with a set of attacks (goals) to search for; the output is either the proof that no attack has been found or an abstract attack trace for each attack found. The algorithm executes symbolically the program graph searching for given goal locations, which in our case represent attacks found on the given scenario of the protocol. In Fig. 8-left, we have depicted a simplified version of a generic program graph, highlighting a location n from which a path leading to a goal location starts. In the case when we fail to reach a goal during a search along an edge (Fig. 8-center), an annotation, i.e., a formula expressing a condition under which no goal can be reached, is produced by using Craig interpolation. Informally speaking, the annotation, $\hat{\imath}$ in the figure, will be a formula implied by (a formula describing the state originated by) the execution *exec*1 and inconsistent with (a formula describing the state reached at) the goal location. Through a backtrack phase, such an annotation is propagated to the preceding nodes of the edge and can be used to block a later phase of symbolic execution along an uninteresting run. Namely, this will happen when the formula describing the state reached by such a later execution (*exec*2 in Fig. 8-right) implies the annotation (where the absence of an annotation can be interpreted as *false*). In such cases, we can in fact foresee that we are in a run that will not reach a goal.

4.1. Preliminary definitions

4.1.1. The annotation language

In what follows, we use a *two-sorted first-order logic with equality*, in which the graph annotations will be expressed. The signature of the first sort is based on the algebra of messages defined in Section 2,

over which we also allow a set of unary predicates DY^j_{IK} for $1 \leqslant j \leqslant n$ with a fixed $n \in \mathbb{N}$, whose meaning will be clarified below, and a ternary predicate *witness*. The signature of the second sort contains a set of variables (denoted in our examples by X possibly subscripted) and uninterpreted constants (for which we use integers as labels), and allows no functions and no predicates other than equality. We assume fixed the sets of constants and denote by $\mathcal{L}(\mathcal{V})$ the *set of well-formed formulas* of such a two-sorted first-order language defined over a (also two-sorted) set $\mathcal{V}$ of variables, which we will instantiate with the concrete program variables of our SiL programs. For what concerns the semantics, the domain of the discourse will be the set of possible data values of *SiL*. *SiL* data states, which are ultimately variable assignments, can be seen as models.

4.1.2. Symbolic execution notions

Before presenting the algorithm, we introduce some notions concerning symbolic execution. In the following, we will assume given a program graph (Λ, l_0, Δ).

Definition 16. Let V be the set of program variables. A *symbolic data state* is a triple (P,C,E), where P is a (again, two-sorted) *set of parameters*, i.e., variables not in V, $C \in \mathcal{L}(P)$ is a *constraint over the parameters*, and the *environment* E is a map from the program variables V to terms of the corresponding sort defined over P, where, in particular, *IK* is mapped to a set of message terms and *witness* to a set of triples of message terms. We write Ξ to denote the *set of symbolic data states*. □

Intuitively, a symbolic data state ξ represents a set of "concrete" (SiL) data states parametrically and it can be characterized by the formula

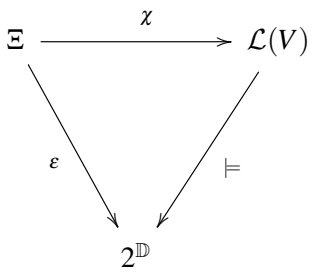
$$\chi(\xi) = C \wedge (\bigwedge_{v \in V \setminus \{IK, witness, attack\}} (v = E(v))) \wedge (\bigwedge_{m \in E(IK)} DY^0_{IK}(m)) \wedge (\bigwedge_{(m_1,m_2,m_3) \in E(witness)} witness(m_1,m_2,m_3)) \wedge \bigwedge_{E(attack)=true} attack.$$

Note that the variable *IK* is treated in a particular way, i.e., we translate the fact that $E(IK) = M$ for some set M of parametric messages into a formula expressing that a predicate DY^0_{IK} holds for all the messages in M.

Note also that E assigns a value (a term) to the program variables, but not to the parameters. It follows that we can associate to each symbolic data state ξ the set $\varepsilon(\xi)$ of all the (concrete) data states obtained from ξ by considering any valuation of the parameters that satisfies the constraint in $\chi(\xi)$, i.e.,

$$\varepsilon(\xi) = \{\varsigma \in \mathbb{D} \mid \varsigma \models \exists P \cdot \chi(\xi)\} .$$

That is, a symbolic data state is connected, via ε, to a set of concrete data states Γ and, via χ, to a first-order formula in $\mathcal{L}(V)$; in turn, the models of such a formula are all the concrete data states in Γ. The relationship between the mentioned notions can be summarized by means of the following diagram:



We assume a defined initial symbolic data state ξ_0 for which $\varepsilon(\xi_0) = \{d_0\}$ (in this case, we have only one concrete data state, as we can assume that the set of parameters is empty for ξ_0).

Definition 17. A *symbolic state* is a pair $(l, \xi) \in \Lambda \times \Xi$. A *symbolic interpreter* $SI : \mathcal{A} \to \Xi \to \Xi$, where $\mathcal{A}$ is the set of SiL actions, is a total map such that for each symbolic data state ξ and action a, we have $\varepsilon(SI(a)(\xi)) = \{\varsigma \in \mathbb{D} \mid < a, \varsigma' \Downarrow \varsigma >, \varsigma' \in \varepsilon(\xi)\}$. □

Intuitively, *SI* takes an action a and a symbolic data state ξ and returns a symbolic data state, which represents the set of (concrete) data states obtained by executing the action a on $\varepsilon(\xi)$.

The previous definitions do not define explicitly a symbolic interpreter, but only specify that it has to satisfy some conditions on the semantics of *SiL*. It is however not difficult to see that such a symbolic interpreter indeed exists and can be easily defined constructively. This is in fact done concretely in our implementation. We start with an empty set of parameters, an empty set of constraints and an empty environment. Assignments modify the environment in a way that is specified by the operational semantics of *SiL* (just consider that a value can now also be parametrical). Conditions in an *if-statement*, which typically involve variables X_i or Y_i, modify the constraint C, represented in the implementation as a set of equalities and predicates of the form $IK \vdash M$. When a new variable X_i or Y_i is introduced in a conditional, an equality between the variable and the corresponding parameter is also added in the environment. For instance, let us consider the statement

```
4  if (IK |- {S2_Bob.Y_1,S2_Bob.Y_2}_pk(S2_Bob.Actor))
5    then S2_Bob.Na := S2_Bob.Y_1;
6         S2_Bob.A := S2_Bob.Y_2;
7    else end;
```

of Program 3 in Example 7. The symbolic execution of the conditional will consist in adding the pairs $(S2\_Bob.Y\_1, y_1)$ and $(S2\_Bob.Y\_2, y_2)$ to the environment, and the predicate $IK \vdash \{y_1, y_2\}_{pk(b)}$ to the constraint (we are assuming here that b is the value currently associated to $S2\_Bob.Actor$ in the environment, while y_1 and y_2 are newly introduced parameters). The symbolic execution of the *then* branch further updates the environment by adding to it the pairs $(S2\_Bob.Na, y_1)$ and $(S2\_Bob.A, y_2)$. These steps correspond to steps $4-6$ of Example 9, in which further details concerning our construction of a symbolic interpreter will be presented.

4.1.3. IntraLA basic notions

Definition 18. An *algorithm state* is a triple (Q, A, G), where Q is the set of *queries* (where a query is a symbolic state), A is a *program annotation* (or simply annotation, for short) and $G \subseteq \Lambda$ is the set of *goal locations* that have not been reached. □

During the execution of the algorithm, the set of queries is used to keep track of which symbolic states still need to be considered, i.e., of those symbolic states whose location has at least one outgoing edge that has not been symbolically executed, and the annotation is a decoration of the graph used to prune the search. Formally:

Definition 19. A *program annotation* is a set of pairs in $(\Lambda \cup \Delta) \times \mathcal{L}(V)$. We will write these pairs in the form $l : \phi$ or $e : \phi$, where l is a location, e is an edge and ϕ is a formula called the *label*. We define $A(el) = \bigvee\{\phi \mid el : \phi \in A\}$ for el an edge or a location. □

We note here that an empty set of annotations $A(el) = \emptyset$ evaluates to false.

Definition 20. For an edge $e = (l_h, a, l_{h+1})$, the *label* $e : \phi$ *is justified in* A whenever for each data state ς_1 such that $\varsigma_1 \models \phi$ and $< a, \varsigma_1 > \Downarrow \varsigma_2$, we have $\varsigma_2 \models A(l_{h+1})$. In that case, we write $\mathcal{J}(e : \phi, A)$.

Let $Out(l)$ be the set of outgoing edges from a location l. The *label* $l : \phi$ *is justified in* A when, for all edges $e \in Out(l)$, there exists $e : \psi \in A$ such that ψ is a logical consequence of ϕ.

An *annotation is justified* when all its elements are justified. □

A justified annotation is inductive and if it is initially justified, then it is an inductive invariant. The algorithm maintains the invariant that A is always justified.

Definition 21. A *query* $q = (l, \xi)$ *is blocked* by a formula ϕ when $\varsigma \models \phi$ for each $\varsigma \in \varepsilon(\xi)$ and we then write $Bloc(q, \phi)$.

The *edge* e *is blocking* the query q when $Bloc(q, A(e))$ and the *location* l *is blocking* the query q when $Bloc(q, A(l))$. □

The algorithm also maintains, as invariants, the facts that no symbolic state (i.e, no query) in Q is blocked and that for all goals l in G, we have $A(l) = false$.

4.2. The rules of our algorithm

The rules of our algorithm are given in Fig. 9.

4.2.1. Initialization

The first rule applied is always *Init*, which initializes the algorithm state, i.e., the algorithm starts from the initial location, the initial symbolic data state, an empty annotation and a set G_0 of goals to search for, which is given as input together with the graph. After the application of *Init*, the rules *Decide*, *Learn* and *Conjoin* can be applied whenever their side-conditions are satisfied.

4.2.2. Symbolic execution steps

The *Decide* rule is used to perform symbolic execution. By symbolically executing one program action, it generates a new query (l_{h+1}, ξ_{h+1}) from an existing one $(q = (l_h, \xi_h))$. It may choose any edge that is not blocking the query q and the symbolic successor state generated by the action a on such an edge. If this generated query is itself not blocked, it is added to the query set.

4.2.3. Backtracking steps

When the symbolic execution using the *Decide* rule gets blocked, two rules are used for backtracking:

(i) *Conjoin*, which merges annotations coming from distinct branches; and
(ii) *Learn*, which generates annotations.

The rule *Conjoin* is used when all the outgoing edges of the location l_h (in a query q) are blocking q. The rule blocks the query q by labeling its location with the conjunction of the labels that block the outgoing edges. If the location is a goal, then we can remove it from the set of remaining goals. Moreover, the query is discarded from the set q.

Finally, if some outgoing edge $e = (l_h, a, l_{h+1})$ is not blocking the query q, but the symbolic step defined by *SI* along that edge leads to a query blocked by $A(l_{h+1})$, then the rule *Learn* is applied. Namely, this is the case when the application of *SI* on ξ_h, with respect to the action a, would result in a symbolic data state ξ_{h+1} such that each model in $\varepsilon(\xi_{h+1})$ satisfies $A(l_{h+1})$. In particular, when the annotation $A(l_{h+1})$ of the location to be reached is *false*, as it is the case when a location is encountered for the first time, the rule is applied if $\varepsilon(\xi_{h+1})$ is empty, i.e., $\chi(\xi_{h+1})$ is unsatisfiable.

INITIALIZATION

$$\frac{}{\{(l_0,\xi_0)\},\emptyset,G_0}\ \mathit{Init}$$

SYMBOLIC EXECUTION STEPS

$$\frac{\begin{array}{r} q=(l_h,\xi_h)\in Q \\ e=(l_h,a,l_{h+1})\in \Delta \\ \neg \mathit{Bloc}(q,A(e)) \\ \xi_{h+1}=SI(a)(\xi_h) \\ \neg \mathit{Bloc}((l_{h+1},\xi_{h+1}),A(l_{h+1})) \end{array}}{Q,A,G \over Q+(l_{h+1},\xi_{h+1}),A,G}\ \mathit{Decide}$$

BACKTRACKING STEPS

$$\frac{\begin{array}{r} q=(l_h,\xi_h)\in Q \\ \neg \mathit{Bloc}(q,A(l_h)) \\ (\forall e\in \mathit{Out}(l_h).e:\phi_e\in A\wedge \mathit{Bloc}(q,\phi_e)) \\ \phi=\bigwedge\{\phi_e \mid e\in \mathit{Out}(l_h)\} \end{array}}{Q,A,G \over Q-q,A+l_h:\phi,G-l_h}\ \mathit{Conjoin}$$

$$\frac{\begin{array}{r} q=(l_h,\xi_h)\in Q \\ e=(l_h,a,l_{h+1})\in \Delta \\ \neg \mathit{Bloc}(q,A(e)) \\ \mathit{Bloc}(q,\phi) \\ \mathcal{J}(e:\phi,A) \end{array}}{Q,A,G \over Q,A+e:\phi,G}\ \mathit{Learn}$$

Fig. 9. Rules of the algorithm IntraLA with corresponding side conditions. Intuitively, the subscripts h and $h+1$ in l/ξ represent the current and successive location/state respectively.

The *Learn* rule then infers a new label ϕ that blocks the edge, where the formula ϕ can be any formula that both blocks the current query and is justified. We note that the fact that there exists such a formula ϕ implies that the action a leads indeed to a blocked symbolic state, which is the reason why we do not need to include this condition among the side-conditions of the rule. In fact, by the definition of blocked query, $\mathit{Bloc}(q,\phi)$ implies $\varsigma \models \phi$ for each $\varsigma \in \varepsilon(\xi_h)$. Furthermore, by the definition of a justified label, $\mathcal{J}(e:\phi,A)$ implies that for each data state ς such that $\varsigma \models \phi$ and $<a,\varsigma>\Downarrow \varsigma'$, we have $\varsigma' \models A(l_{h+1})$. It follows that, given $\xi_{h+1}=SI(a)(\xi_h)$, for each $\varsigma' \in \varepsilon(\xi_{h+1})$, we have $\varsigma' \models A(l_{h+1})$, i.e., (l_{h+1},ξ_{h+1}) is blocked by $A(l_{h+1})$.

In Section 4.3, we will explain how the formula ϕ can be obtained by exploiting the Craig interpolation lemma.

4.3. The generation of interpolants

We have seen in Section 4.2 that the rule *Learn* (Fig. 9) requires the generation of a formula ϕ that blocks the current query and is justified, to be used as an annotation. This can be obtained by using the Craig interpolation lemma [19], which states that given two first-order formulas α and β such that $\alpha \wedge \beta$ is inconsistent, there exists a formula ϕ (their *interpolant*) such that α implies ϕ, ϕ implies $\neg\beta$ and $\phi \in \mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$, where for a formula γ, $\mathcal{L}(\gamma)$ denotes the first-order language defined over the uninterpreted symbols occurring in γ.

We will first introduce some notions concerning the description of data states and actions in our annotation language and then describe how to obtain, in our case, the formula ϕ as an appropriate interpolant.

Let μ be a term, a formula, or a set of terms or of formulas. We write μ' for the result of adding one prime to all the non-logical symbols in μ. Intuitively, v' refers to the value of the variable v in the target state of a transition. It is used in *transition formulas*, i.e., formulas in $\mathcal{L}(V \cup V')$. Since the semantics of a SiL action (see Section 3.1) expresses how we move from a data state to another, we can easily associate to it a transition formula. In the following, we will write $Sem(a)$ to denote the transition formula corresponding to the action a. For example, the semantics of the assignment of a constant c to a variable V ($Sem(V := c)$) is $V' = c$.

In the context of our graphs, the most interesting case is when the action a is represented by a conditional statement, with a condition of the form $IK \vdash M$ for some message M, which intuitively means that the message M can be derived from a set of messages IK by using the rules of $\mathcal{N}_{DY}$ of Fig. 1. In our treatment, we fix a value n as the maximum number of inference steps that the intruder can execute in order to derive M. This is a limitation of our method, which, as we already remarked in Section 2.1, is however mitigated by several results (e.g., [9]) that show that, when terms are interpreted in the free algebra and a finite number of sessions are considered, as in our case, it is indeed possible to set an upper bound on the number of inference steps needed. Such a value can be established a priori by observing the set of messages exchanged along the protocol scenario; we assume such an n to be fixed for the whole scenario.[4]

We use formulas of the form $DY^j_{IK}(M)$, for $0 \leqslant j \leqslant n$, with the intended meaning that M can be derived in j steps of inference by using the rules of $\mathcal{N}_{DY}$. In particular, the predicate DY^0_{IK} is used to represent the initial knowledge IK, before any inference step is performed. Under the assumption on the n mentioned above, the statement $IK \vdash M$ can be expressed in our language as the formula $DY^n_{IK}(M)$.

[4]The ability of the intruder of generating new messages can be simulated by enriching his initial knowledge with a set of constants not occurring elsewhere in the protocol specification. Since we consider finite scenarios, the size of such a set can also be bounded a priori.

The formula

$$\begin{aligned}\varphi_j = \forall M.\,(DY^{j+1}_{IK}(M) \leftrightarrow \Big(& DY^{j}_{IK}(M)\\ & \vee (\exists M'.\, DY^{j}_{IK}([M,M']) \vee DY^{j}_{IK}([M',M]))\\ & \vee (\exists M_1,M_2.\, M=[M_1,M_2] \wedge DY^{j}_{IK}(M_1) \wedge DY^{j}_{IK}(M_2))\\ & \vee (\exists M_1,M_2.\, M=\{M_1\}_{M_2} \wedge DY^{j}_{IK}(M_1) \wedge DY^{j}_{IK}(M_2))\\ & \vee (\exists M'.\, DY^{j}_{IK}(\{M\}_{M'}) \wedge DY^{j}_{IK}(inv(M'))\\ & \vee (\exists M'.\, DY^{j}_{IK}(\{M\}_{inv(M')}) \wedge DY^{j}_{IK}(M'))\\ & \vee (\exists M_1,M_2.\, M=\{|M_1|\}_{M_2} \wedge DY^{j}_{IK}(M_1) \wedge DY^{j}_{IK}(M_2))\\ & \vee (\exists M'.\, DY^{j}_{IK}(\{|M|\}_{M'}) \wedge DY^{j}_{IK}(M'))\Big),\end{aligned}$$

in which $\leftrightarrow$ denotes the double implication and every quantification has to be intended over the sort of messages, expresses (as a disjunction) all the ways in which a given message can be obtained by the intruder in one inference step, i.e., by a single application of one of the rules in the system $\mathcal{N}_{DY}$, thus moving from a knowledge (denoted by the predicate) DY^{j}_{IK} to a knowledge (denoted by the predicate) DY^{j+1}_{IK}.

A theory $\mathcal{T}_{Msg}(n)$ over the sort of messages is obtained by enriching classical first-order logic with equality with the axioms φ_j, for $1 \leqslant j < n$, together with an additional set of axioms that formalize that in the free algebra of messages any two distinct ground terms are not equal, e.g., $\forall M_1.M_2.M_3.M_4.\,([M_1,M_2] \neq \{M_3\}_{M_4})$.

Our translation of the program statement $IK \vdash M$ into the formula $DY^{n}_{IK}(M)$ is justified by the following result. This is proved by induction on the *height of a derivation tree* Π in the system $DY(IK)$, which is defined as the greatest number of successive applications of rules in Π.

Theorem 2. *Let M be a ground message, $n \in \mathbb{N}$, IK a set of ground messages and $\mathcal{I}$ an interpretation of $\mathcal{T}_{Msg}(n)$ such that $IK = \mathcal{I}(DY^{0}_{IK})$. Then $\mathcal{I}$ satisfies the formula $DY^{n}_{IK}(M)$ iff there exists a derivation of $M \in DY(IK)$ of height at most $n+1$ in the system $\mathcal{N}_{DY}$.*

Proof. ($\Rightarrow$) Assume that the interpretation $\mathcal{I}$ satisfies the formula $DY^{n}_{IK}(M)$, denoted $\mathcal{I} \models DY^{n}_{IK}(M)$. We proceed by induction on n. If $n = 0$, then we have $\mathcal{I} \models DY^{0}_{IK}(M)$, i.e., $M \in \mathcal{I}(DY^{0}_{IK})$ which by hypothesis gives $M \in IK$. But then there exists a derivation in $\mathcal{N}_{DY}$ of $M \in DY(IK)$, obtained by a single application of the rule G_{axiom}. Now assume we have proved the assertion for $n = j-1$ and consider $n = j$. Since $\mathcal{I}$ satisfies the premise of the left-to-right implication in φ_{j-1}, i.e., $DY^{j}_{IK}(M)$, then it must also satisfy one of the disjuncts in the conclusion. We have a case for each disjunct. We consider two of them; the others are similar. (*i*) Let $\mathcal{I} \models DY^{j-1}_{IK}(M)$. By induction hypothesis, there exists a derivation of $M \in DY(IK)$ in $\mathcal{N}_{DY}$ of height at most j, which is the derivation we were looking for. (*ii*) Let $\mathcal{I} \models \exists M'.\, DY^{j-1}_{IK}([M,M']) \vee DY^{j-1}_{IK}([M',M])$. We can assume there exists a message M' such that $\mathcal{I} \models DY^{j-1}_{IK}([M,M'])$ (the other case is symmetrical). By induction hypothesis, there exists a derivation of $[M,M'] \in DY(IK)$ in $\mathcal{N}_{DY}$ of height at most j. A further application of A_{pair_i} gives a derivation of $M \in DY(IK)$ of height at most $j+1$.

($\Leftarrow$) Again, we proceed by induction on n. If $n = 0$, the only admissible derivation of $M \in DY(IK)$ is the one given by an application of G_{axiom}. It follows that $M \in IK$. Then $IK = \mathcal{I}(DY^{0}_{IK})$ implies $\mathcal{I} \models DY^{0}_{IK}(M)$. Now let us consider $n = j$ and assume we have a derivation of $M \in DY(IK)$ of length at most

$j+1$. Let r be the last rule applied. We have one case for each rule in $\mathcal{N}_{DY}$. Let r be G_{pair}. It follows that we have two derivations, of length at most j, of $M_1 \in DY(IK)$ and $M_2 \in DY(IK)$, respectively, where $M = [M_1, M_2]$. By induction hypothesis, we have $\mathcal{I} \models DY_{IK}^{j-1}(M_1)$ and $\mathcal{I} \models DY_{IK}^{j-1}(M_2)$, which implies that $\mathcal{I}$ satisfies one of the disjuncts in the premise of the right-to-left implication of φ_{j-1}. It follows that its conclusion must also be satisfied, i.e., $\mathcal{I} \models DY_{IK}^{j}(M)$. The other cases can be treated similarly. □

Now let $\alpha = \chi(\xi_h)$ and $\beta = Sem(a) \wedge \neg A(l_{h+1})'$. Then we can obtain the formula ϕ we are looking for, during an application of the rule *Learn*, as an interpolant for α and β, possibly by using an interpolating theorem prover. With regard to this, we observe that, in the presence of our finite scenario assumption, when mechanizing such a search, the problem can be simplified by restricting the domain to a finite set of messages.

4.4. Output and correctness of the algorithm

The algorithm terminates when no rules can be applied, which implies that the query set is empty. We note that the algorithm always terminates (after a full exploration of the paths in the program graph, in the worst case) as it is just an optimization over the standard symbolic execution algorithm. In [15], the correctness of the algorithm, with respect to the goal search, is proved: the proof given there applies straightforwardly for the slightly simplified version we have given here.

Theorem 3. *Let G_0 be the set of goal locations provided in input. If the algorithm terminates with the algorithm state $(\emptyset, A, G)$, then all the locations in $G_0 \setminus G$ are reachable and all the locations in G are unreachable.*

The output of our method can be of two types. If no goal has been reached, i.e., $G_0 \subseteq G$, then we have a proof of the fact that no attack can be found, with respect to the security property of interest, in the finite scenario that we are considering. Otherwise, for each reachable goal location, i.e., in $G_0 \setminus G$, we can generate an abstract attack trace. We also note that, by a trivial modification of the rule *Conjoin*, we can easily obtain an algorithm that keeps searching for a given goal even when this has already been reached through a different path, thus allowing for extracting more attack traces for the same goal on a given scenario.

Such traces can be inferred from the information deducible from the symbolic data state (P, C, E) corresponding to the last step of execution. We proceed as follows. First of all, we can reconstruct the order in which sessions have been interleaved. This information is given by the value of the parameters corresponding to the variables X_j, for j an integer, which are specified in the constraint C. This allows us to obtain the sequence of messages exchanged, expressed in terms of program variables. Then, by using the maps in E, each such a variable can be associated to a function over the set of parameters P, and possibly further specified by the constraints over the parameters in C. It follows that the final result will be a sequence of messages where all the variables have been replaced by (functions over) parameters. Such a sequence constitutes our attack trace. In the case when the value of some parameter is not fully specified by the conditions in C, we have a parametrical attack trace, which can be instantiated in more than one way. A concrete example of this can be found in Example 9.

Example 9. *We continue our running example by showing the execution of the algorithm on some interesting paths of the graph defined in Section 3.2 for the protocol NSL: Table 1 summarizes the algorithm execution.*

Table 1: Execution of the algorithm on the program graph for the protocol NSL.

#	R	**Query**	**Edge**	Q	A	C	E
0	I	(l_0, s_0)	-	l_0, ξ_0	$\emptyset$	$\emptyset$	$\emptyset$
1	D	(l_0, ξ_0)	(l_0, l_1)	$(l_0, \xi_0), (l_1, \xi_1)$	$\emptyset$	C_0	$E_0 \oplus \{(S1\_Alice.Actor, a), (S1\_Alice.B, i), (S1\_Alice.Na, c_0), (S2\_Alice.Actor, a), (S2\_Alice.B, b), (S2\_Alice.Na, c_1), (S2\_Bob.A, a), (S2\_Bob.Actor, b), (IK, \{a, b, i, pk(a), pk(b), pk(i), inv(pk(i))\})\}$
2	D	(l_1, ξ_1)	(l_1, l_2)	$Q_1 \cup \{(l_2, \xi_2)\}$	$\emptyset$	C_1	$E_1 \oplus \{(IK, IK_1 \cup \{\{c_1, a\}_{pk(b)}, \{c_0, a\}_{pk(b)}\})\}$
3	D	(l_2, ξ_2)	(l_2, l_3)	$Q_2 \cup \{(l_3, \xi_3)\}$	$\emptyset$	$C_2 \cup \{(x_1 = 3)\}$	$E_2 \oplus \{(X_1, x_1)\}$
4	D	(l_3, ξ_3)	(l_3, l_4)	$Q_3 \cup \{(l_4, \xi_4)\}$	$\emptyset$	$C_3 \cup \{IK_2 \vdash \{y_1, y_2\}_{pk(b)}\}$	$E_3 \oplus \{(S2\_Bob.Y\_1, y_1), (S2\_Bob.Y\_2, y_2)\}$
5	D	(l_4, ξ_4)	(l_4, l_5)	$Q_4 \cup \{(l_5, \xi_5)\}$	$\emptyset$	C_4	$E_4 \oplus \{(S2\_Bob.Na, y_1)\}$
6	D	(l_5, ξ_5)	(l_5, l_6)	$Q_5 \cup \{(l_6, \xi_6)\}$	$\emptyset$	C_5	$E_5 \oplus \{(S2\_Bob.A, y_2)\}$
7	D	(l_6, ξ_6)	(l_6, l_7)	$Q_6 \cup \{(l_7, \xi_7)\}$	$\emptyset$	C_6	$E_6 \oplus \{(S2\_Bob.Nb, c_2)\}$
8	D	(l_7, ξ_7)	(l_7, l_8)	$Q_7 \cup \{(l_8, \xi_8)\}$	$\emptyset$	C_7	$E_7 \oplus \{(IK, IK_7 \cup \{\{y_1, c_2, b\}_{pk(y_2)}\})\}$
9	D	(l_8, ξ_8)	(l_8, l_9)	$Q_8 \cup \{(l_9, \xi_9)\}$	$\emptyset$	$C_8 \cup \{(x_{11} = 2)\}$	$E_8 \oplus \{(X_{11}, x_{11})\}$
10	D	(l_9, ξ_9)	(l_9, l_{10})	$Q_9 \cup \{(l_{10}, \xi_{10})\}$	$\emptyset$	$C_9 \cup \{IK_8 \vdash \{c_1, y_1, b\}_{pk(a)}\}$	$E_9 \oplus \{(S1\_Alice.Y\_1, y_4)\}$
11	D	(l_{10}, ξ_{10})	(l_{10}, l_{11})	$Q_{10} \cup \{(l_{11}, \xi_{11})\}$	$\emptyset$	C_{10}	$E_{10} \oplus \{(S2\_Alice.Nb, y_3), (S2\_Alice.Y_3, y_3)\}$
12	D	(l_{11}, ξ_{11})	(l_{11}, l_{12})	$Q_{11} \cup \{(l_{12}, \xi_{12})\}$	$\emptyset$	C_{11}	$E_{11} \oplus \{(IK, IK_{11} \cup \{\{y_3\}_{pk(b)}\})\}$
13	D	(l_{12}, ξ_{12})	(l_{12}, l_{13})	$Q_{12} \cup \{(l_{13}, \xi_{13})\}$	$\emptyset$	C_{12}	$E_{12} \oplus \{(witness(a, b, \{y_3\}_{pk(b)}), true)\}$
14	D	(l_{13}, ξ_{13})	(l_{13}, l_{14})	$Q_{13} \cup \{(l_{14}, \xi_{14})\}$	$\emptyset$	$C_{13} \cup \{(x_9 = 1)\}$	$E_{13} \oplus \{(X_9, x_9)\}$
15	D	(l_{14}, ξ_{14})	(l_{14}, l_{15})	$Q_{14} \cup \{(l_{15}, \xi_{15})\}$	$\emptyset$	$C_{14} \cup \{IK_{13} \vdash \{c_0, y_4, i\}_{pk(a)}\}$	E_{14}
16	D	(l_{15}, ξ_{15})	(l_{15}, l_{16})	$Q_{15} \cup \{(l_{16}, \xi_{16})\}$	$\emptyset$	C_{15}	$E_{15} \oplus \{(S1\_Alice.Nb, y_4)\}$
17	D	(l_{16}, ξ_{16})	(l_{16}, l_{17})	$Q_{16} \cup \{(l_{17}, \xi_{17})\}$	$\emptyset$	C_{16}	$E_{16} \oplus \{(IK, IK_{16} \cup \{\{y_4\}_{pk(i)}\})\}$
18	D	(l_{17}, ξ_{17})	(l_{17}, l_{18})	$Q_{17} \cup \{(l_{18}, \xi_{18})\}$	$\emptyset$	C_{17}	$E_{17} \oplus \{(witness(a, i, \{y_4\}_{pk(i)}), true)\}$
19	D	(l_{18}, ξ_{18})	(l_{18}, l_{19})	$Q_{18} \cup \{(l_{19}, \xi_{19})\}$	$\emptyset$	$C_{18} \cup \{IK_{18} \vdash \{c_2\}_{pk(b)}\}$	E_{18}
20	L	(l_{19}, ξ_{19})	-	Q_{19}	$(l_{19}, l_{20}) : S2\_Bob.A = i$	C_{19}	E_{19}
21	C	(l_{19}, ξ_{19})	(l_{19}, l_{20})	Q_{18}	$A_{20} \cup \{l_{19} : S2\_Bob.A = i\}$	C_{20}	E_{20}
22	L	(l_{18}, ξ_{18})	-	Q_{18}	$A_{21} \cup \{(l_{18}, l_{19}) : S2\_Bob.A = i\}$	C_{21}	E_{21}
23	C	(l_{18}, ξ_{18})	(l_{18}, l_{19})	Q_{17}	$A_{22} \cup \{l_{18} : S2\_Bob.A = i\}$	C_{22}	E_{22}
...							
33	C	(l_{14}, ξ_{14})	(l_{14}, l_{15})	Q_{27}	$A_{32} \cup \{l_{14} : S2\_Bob.A = i\}$	C_{32}	E_{32}

For readability, we have not reported the evolution of parameters and goals set. We remark that each new parameter is added to the parameters set once used and the goals set is initialized with the goal locations corresponding to the translation of the authentication goal auth (see Section 3.2 for details) but, given that no goal is reached, the goals set does not change during the execution of the algorithm. Note that in the table we use statements of the form $IK \vdash M$ in the constraint set as an abbreviation for the formulas over the parameters that make the (translation of the) statement satisfiable, according to the definition above. Q_i, C_i and E_i denote, respectively, the set of queries, the set of constraints and the environment at step i of the execution. We have also used # to indicate the step number and R to indicate which rule is applied.

The first path we show (summarized by the message sequence chart in Fig. 10) reaches a goal location with an unsatisfiable state and then annotates it with an interpolant, while the other ones reach the previously annotated path and then block their executions (thus saving some execution steps). The algorithm starts, as described in Table 1, by using the Init rule to initialize the algorithm state and then it symbolically executes the program graph from query (l_0, ξ_0) to (l_{18}, ξ_{18}) using the Decide rule (steps 0–19*). For readability, in Tab. 1 and Fig. 10, all the variables (along with IK) are initialized in location l_0.*

In step 20*, the algorithm blocks its symbolic execution because the edge (l_{19}, l_{20}) is labeled with the goal action for an* authentication goal *and any possible symbolic execution step leads to a blocked symbolic data state (i.e., the location reached has no other outgoing edges).*

We now show how the algorithm calculates an interpolant and how it is propagated annotating the graph (to prevent the execution of paths that will not reach a goal location). Afterwards, we discuss how the constraints imposed by the interpolant translate to the NSL protocol and why it prevents the executions of paths that would not reach the goal location.

The backtrack phase starts and, until step 33*, the algorithm creates interpolants to annotate the program graph and then it propagates annotations up to the location l_{14} (where the symbolic execution restarts with the Decide rule, but we have not shown it in Table 1 for lack of space).*

As shown in Fig. 11, there are two other paths that reach location l_{18}.[5] *Each path that reaches this location has already executed an action of the form $IK \vdash \{N_A, N_B, B\}_{pk(A)}$ (second session where both Alice and Bob are played by honest agents). As described in [7], it is impossible for the DY intruder to create a message of the form $\{N_A, N_B, B\}_{pk(A)}$ from its knowledge (IK) if the intruder is not explicitly playing the role of the sender, i.e., A. Note that, the intruder receives the message $\{N_A, N_B, B\}_{pk(A)}$ but if he does not play the role of A, he can only forward the message (i.e., there is no way for the intruder to know the components N_A and N_B) and this contradicts the witness predicate in the goal (if the intruder forwards all the messages there is no violation of the authentication property).*

This means that each symbolic state that reaches location l_{18} implies the interpolant $S2\_Bob.A = i$. This is a concrete example of how the annotation method can help (and improve) the search procedure: in NSL we can stop following every path that reaches location l_{18} as the annotation method ensures that we will never reach a goal location.

While with NSL the algorithm concludes with no attacks found, if we consider the original protocol NSPK (i.e., remove Lowe's addition of "B" in the second message of the protocol), then our method reaches the goal location with an execution close to the one we have just provided. In fact, in NSPK, when we compute the step after the 19^{th}, the intruder rules lead to the goal with the inequality $S2\_Bob.A \neq i$.

[5] Note that, for readability, we have sequentially enumerated the locations encountered in this example. In particular, the locations $17-20$ of Figure 11 correspond, respectively, to the locations $77, 35-37$ of Figure 7.

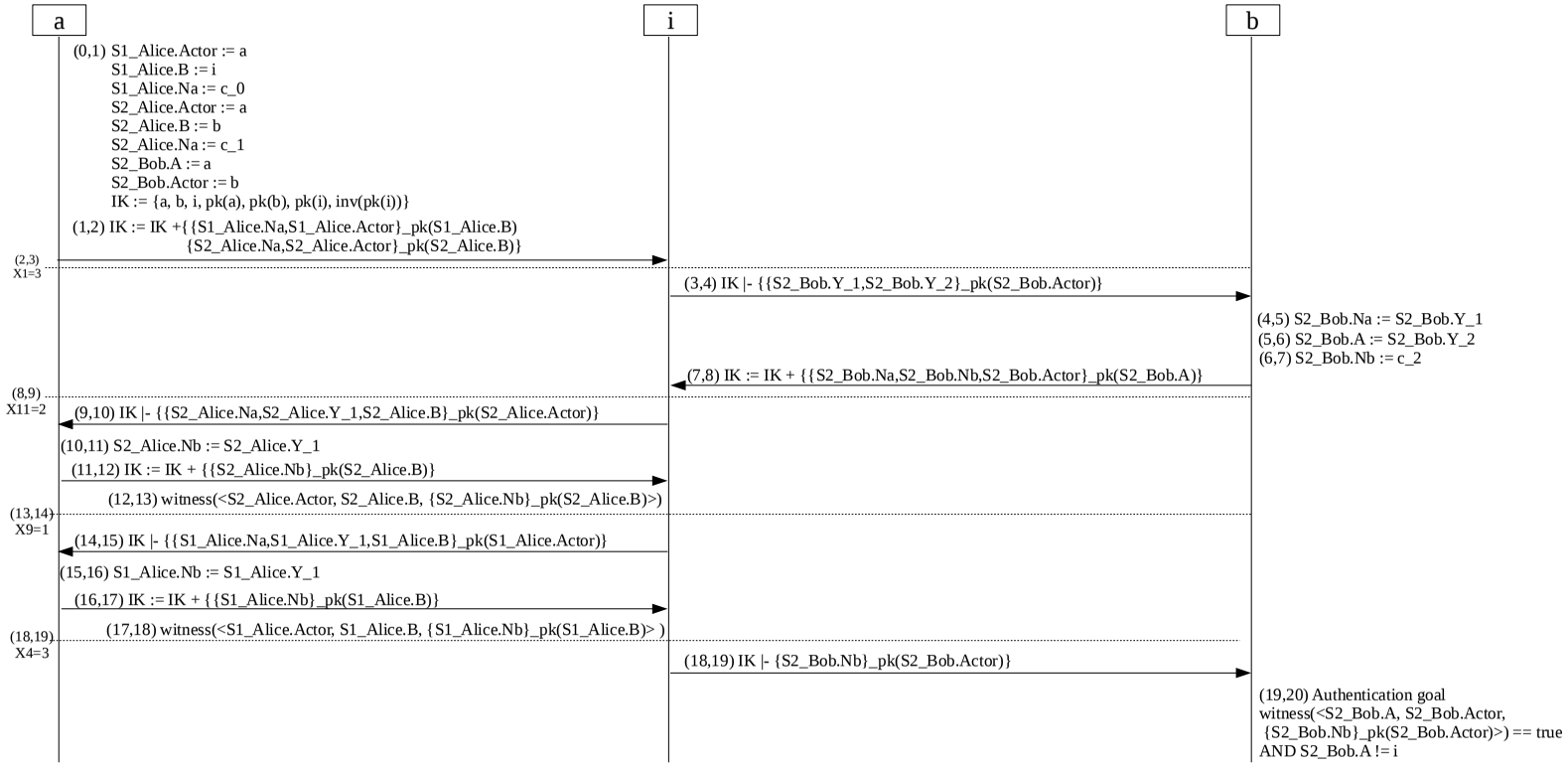


Fig. 10.: Message sequence chart for one execution path of the NSL example. The actions executed in (0,1) and (1,2) have been grouped together for readability.

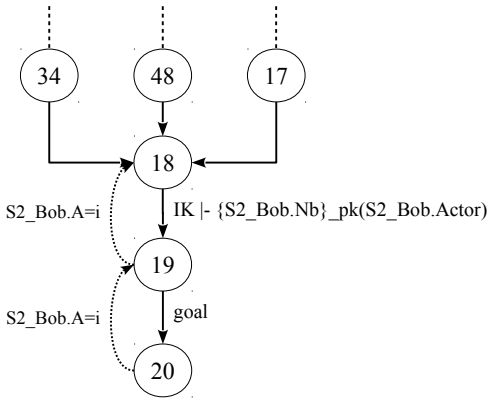


Fig. 11. NSL sub-graph.

This is because the intruder i can perform a man-in-the-middle attack using the initiator entity of the first session in order to decrypt the messages that the receiver sends to i in the second one [7]. To show the attack trace, we first check the path that is used during the algorithm execution to reach the goal location and that is represented by the values of X_j parameters contained in the C_{19} set. In this case, $\{X_{11} = 2, X_9 = 1\} \subseteq C_{19}$, which produces the symbolic attack trace (at state 19 of the algorithm execution) shown in the middle of Fig. 2.

Now, by using the information in ξ_{19}, we can instantiate this trace using parameter and constant values, and thus obtain the instantiated attack trace shown on the right of Fig. 2. We can note from IK_{19} that Y_2 has no constraints on the fact that it has to be i, i.e., the intruder acts as if it were an honest agent (under his real name) in the first session, and then we write the concretization as $i(a)$ to show that the intruder is acting as the honest agent a in the second session and this makes the man-in-the-middle attack possible.

It is also not difficult to extract from this instantiated attack trace a test case, which can then be applied to test the actual protocol implementation. In fact, the constraint set contains a sequence of equalities of the form $X_i = n$, which specify the session to be followed at each branch of the executed path. □

5. The SPiM tool

In order to show that our method concretely speeds up the validation, we have implemented a Java prototype called *SPiM* (*Security Protocol interpolation Method*), which is available at http://regis.di.univr.it/spim.php. As shown in Fig. 12, SPiM takes an ASLan++ specification as input that is automatically translated into a SiL program graph by the translator *ASLan++2Sil*. The program graph is then given as input to the *Verification Engine (VE)*, which verifies the protocol by searching for goal locations that represent attacks on the protocol. The VE is composed of three main components:

(i) a *quantifier elimination* module,
(ii) *DY intruder* and *EUF (Equalities and Uninterpreted Functions) theories* and
(iii) the tools Z3 [30] and iZ3 [31], used for SAT solving and interpolant generation, respectively.

Both Z3 and iZ3 are invoked by *SPiA* (*Security Protocol interpolation Algorithm*), which is our implementation of the algorithm in Section 4. Quantifier elimination and the definition of theories are related

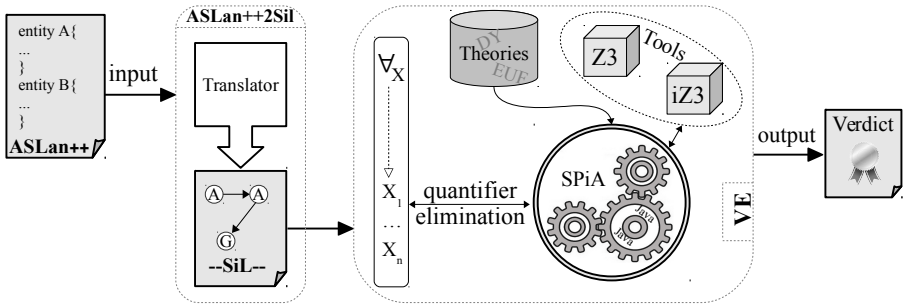


Fig. 12. The SPiM tool.

to the usage of Z3 and iZ3. In fact, as shown in Section 4, our algorithm needs to handle many quantifications and, for performance issues, a module that unfolds each quantifier over the finite set of possible messages has been developed. Moreover, the DY theory has been properly axiomatized (with respect to each formula produced by SPiA) in Z3 and iZ3, which do not support it by default.

More specifically, the VE symbolically executes a program graph. After the execution of an action branching from a node to the next one, it produces a formula, which represents the symbolic state reached. Z3 is then used for a satisfiability check on the newly produced formula. When the symbolic execution of a given path fails to reach a goal, the VE calls iZ3, which generates an annotation (i.e., a formula expressing a condition under which no goal can be reached from the current state) by using Craig's interpolation. By a backtracking phase, SPiA propagates the annotation through the program graph. Such an annotation is possibly used to block a later phase of symbolic execution along an uninteresting run, as explained in Section 4. SPiM concludes reporting either all the different reachable attack states (from which abstract attack traces can be extracted) or that no attack has been found for the given specification.

5.1. Experiments and results

We considered 7 case studies and compared the results obtained by using interpolation-driven exploration (SPiA) and full exploration (*Full-explore*) of the program graph. Full-explore explores the entire graph checking, at each step, if the state is satisfiable or not. If there is an inconsistency, SPiM blocks the execution of the path resuming from the first unexplored path, until it has explored all paths.[6]

Table 2 shows the results obtained (with a general purpose computer), by making explicit the time required for symbolic execution steps (applications of *Decide*) and for interpolant generation (applications of *Learn*). The usage of SPiA has allowed us to speed up the validation (in the context of security protocols, i.e, using the DY intruder) by (i) reducing the number of states to explore and then (ii) lowering the execution time. The relation between (i) and (ii) is due to the fact that the time needed to perform a *Decide* is comparable to the one required to perform a *Learn*, and the time used to propagate the annotations (*Conjoin* rule) is negligible. For example, the time needed to symbolically execute a (sub-)path twice, using Full-explore, is comparable to the time used to execute and annotate the same (sub-)path. But from that point on, if the annotation blocks the execution, only the Full-explore will execute that

[6]It would be possible to modify the Full-explore algorithm and check for inconsistencies at the end of the path instead of at any step but this would lead to an unfair comparison. In fact, a similar improvement could have been implemented also for SPiM, but then it would be difficult to distinguish between the steps pruned by interpolation and those pruned by such an improvement.

Table 2
SPiA vs Full-explore.

Specification (sessions)	**SPiA:** *Decide*+*Learn* **(time)**	**Full-explore:** *Decide* **(time)**	**Speedup %**	**Result**
ISO6 (ab,ab)	311+274 (205m6s)	467* (278m12s)	-26.28 %	no attack found
NSL (ab,ab)	257+234 (57m37s)	631 (173m7s)	-66.71 %	no attack found
NSL (ai,ab)	89+22 (1m30s)	119 (1m49)	-17.43 %	no attack found
NSL (ai,ab, ib)[7]	440+348 (93m51s)	619* (137m23s)	-31.68%	no attack found
NSPK (ab,ab)	257+234 (26m5s)	631 (76m20s)	-65.82 %	no attack found
NSPK (ai,ab)	101+22 (0m56s)	123 (0m51s)	+8.92 %	attack found
Helsinki (ab,ab)	311+274 (112m7s)	660* (261m47s)	-57.17 %	no attack found
Helsinki (ai,ab)	167+88 (13m41)	407 (46m44s)	-70.72 %	attack found

(sub-)graph again. We have observed that, in the case studies analyzed, the annotations block the executions of all those (sub-)paths that do not reach a goal location, thus ensuring a clear improvement of the performances. In particular, when applying a *Decide* moving from a node l_1 to a node l_2, we generate a formula that describes the state of the execution at node l_2 and the axiomatization of the DY theory; this formula is then given to Z3 that "decides" whether it is satisfiable or not. On the other hand, in order to execute a *Learn* between the same S_1 and S_2, we translate the state S_1 with the axiomatized DY theory into a formula α and the semantics of the action a together with all previous annotations into a formula β. In order to find an interpolant we use iZ3 that performs a satisfiability check on the formula $\alpha \wedge \beta$ (very similar to what a *Decide* would do) and from the refutation by resolution steps an interpolant can be calculated in linear time [14, 32]. Finally, the *Conjoin* rule propagates these interpolants without performing other satisfiability checks.

Empirically, the more the program graph grows, the more the annotations prune the search space. This is due to the fact that the number of states pruned by interpolation is usually related to the size of a program graph; this is confirmed by the results in Table 2 and, in particular, by the case studies for which Full-explore has not concluded the execution (marked with an asterisk).

We have also compared the SPiM tool with the three state-of-the-art model checkers for security protocols that are part of the AVANTSSAR platform [1]: CL-AtSe [21], OFMC [3] and SATMC [22].[8] Not surprisingly, Table 3 shows that their average computational times of execution are in general better than ours. This is mainly due to several speed-up techniques implemented by these model checkers and to empirical conditions that can stop the execution (both not implemented yet in SPiM). Table 3 also shows the number of transitions and/or nodes reached during the validations with the exception of SATMC, which does not report them as output. However, for each safe specification (in which no attacks are found), SATMC reached the maximum number of steps (80) permitted as default and the reported timings are comparable to those obtained by SPiM for some specifications; in the case when they are not comparable, it is interesting to observe that SPiM executes a number of rules much higher than 80. For both CL-AtSe and OFMC, on safe specifications, the number of transitions and nodes explored is, in most cases, higher than the number of rules (transitions) of SPiM (Table 2). On unsafe specifications

[7] Evaluation performed to show the scaling behavior for 3 sessions.

[8] For this comparison, given that all these tools support ASLan++, we have used the same input files and we have also used the same general purpose computer used to generate the results in Table 2. We have considered OFMC v2012c, which is the last version that supports ASLan++ although it only supports untyped analysis, while for SATMC and CL-AtSe we have considered versions 3.4 and 2.5-21, respectively, which support typed analysis as SPiM does. Note that the times shown in Table 2 also consider the translation from ASLan++ to SiL program graph (usually several seconds), while in Table 3 we do not show the translation time from ASLan++ to ASLan (the input supported by the three tools), which is usually less than one second.

Table 3
SATMC, CL-AtSe and OFMC.

Specification (sessions)	**SATMC** (v.3.4)	**CL-AtSe** (v.2.5-21)			**OFMC** (v.2012c)		**Result**
	time	transitions	states	time	nodes	time	
ISO6 (ab,ab)	6.318s	452	236	0.034s	8432	3.804s	no attack found
NSL (ab,ab)	14m28s	794	534	0.052s	3236	3.295s	no attack found
NSL (ai,ab)	6m51s	93	69	0.015s	575	0.327s	no attack found
NSPK (ab,ab)	14m10s	794	534	0.053s	8180	3.208s	no attack found
NSPK (ai,ab)	1m56s	14	10	0.014s	96	0.134s	attack found
Helsinki (ab,ab)	7.01s	794	534	0.061s	8180	3.795s	no attack found
Helsinki (ai,ab)	50.8s	14	10	0.017s	96	0.121s	attack found

(where an attack is found), these numbers seem to be in disfavor of SPiM but this is because SATMC, OFMC and CL-AtSe stop their executions once a goal is found, while SPiM searches for every possible attack trace in the program graph (i.e., SPiM features a multi-attack-trace support).

We remark that the aim of SPiM is mainly to show that Craig's interpolation can be used as a speed-up technique also in the context of security protocols and not (yet) to propose an *efficient* implementation of a model checker for security protocol verification. In fact, we do not see our approach as an alternative to such more mature and widespread tools, but we actually expect some interesting and useful interaction. For example, CL-AtSe implements many optimizations, like simplification and rewriting of input specifications, and OFMC implements some optimizations at the intruder level as well as a specific technique, called *constraint differentiation* (CDiff), which considerably prunes the state space (it is more or less equivalent to partial-order reduction techniques typical of model checking, where the reduction is "pushed" to the constraint solving procedure). Moreover, both CL-AtSe and OFMC implement the *step compression* and *protocol simplifications* techniques, which merge together some of the actions performed in the protocol.

We do not see any incompatibility in using interpolation together with such optimization techniques. For instance, CDiff prunes the state space by not considering the same state twice, whereas interpolation works on reducing the search space by excluding some paths during the analysis (i.e., it prunes the execution of some of the paths). Moreover, based on the idea that the intruder controls the network, when the intruder sends a message ($IK \vdash M$) to an honest agent and the honest agent sends back a reply ($IK := IK + \{M\}$), step compression merges the two into a single step. This would reduce the state space but not prevent SPiM from generating and using interpolants.

The only possible side effect that we foresee in using interpolation together with such optimization techniques is that the number of paths pruned by interpolation could decrease when we use it in combination with other techniques. In general, however, although we don't have experimental evidence yet, we expect that if enhanced with such techniques, SPiM could then reach even higher speed-up rates. We are currently working in this direction.

5.2. Analysis of the interpolants generated

The interpolants we have considered so far (with our running example) are quite simplistic for readability reasons. However, the interpolants generated by SPiM can be rather complex formulae (i.e., with hundreds of connectives and variables). In the remainder of this section, in order to give an insight of the kind of information that can occur in an annotation, we describe the details of some of the inter-

Table 4
NSL – SiL path execution

ID	Prog.	Communication	Intruder Action
1	P2	A -> B	$IK := IK + \{S2\_Alice.Na, S2\_Alice.Actor\}\_pk(S2\_Alice.B)$
2	P3	A -> B	$IK \vdash \{S2\_Bob.Na, S2\_Bob.A\}\_pk(S2\_Bob.Actor)$
3	P3	B -> A	$IK := IK + \{S2\_Bob.Na, S2\_Bob.Nb, S2\_Bob.Actor\}\_pk(S2\_Bob.A)$
4	P2	B -> A	$IK \vdash \{S2\_Alice.Na, S2\_Alice.Nb, S2\_Alice.B\}\_pk(S2\_Alice.Actor)$
5	P2	A -> B	$IK := IK + \{S2\_Alice.Nb\}\_pk(S2\_Alice.B)$
6	P3	A -> B	$IK \vdash \{S2\_Bob.Nb\}\_pk(S2\_Bob.Actor)$
7	P3	goal: $S2\_Bob.A \neq i \wedge$ $witness(S2\_Bob.A \text{ -> } S2\_Bob.Actor : \{S2\_Bob.Nb\}\_pk(S2\_Bob.B))$	

polants generated during the execution of SPiA on the running example. Specifically, an interpolant can be composed of two different types of constraints:

- constraints over the knowledge of the intruder; and
- constraints over the instantiation of variables (e.g., constraining session instantiations).

Before going into the details of the interpolants, we recall that in the running example we have considered two sessions:

- Session 1: $Alice = a$, $Bob = i$
- Session 2: $Alice = a$, $Bob = b$

Note that when we generate the SiL graph, we consider one program for each role in each session, but we don't consider programs for the entities played by the intruder. Therefore, we combine three different programs, one for the first session (i.e., considering *Alice* and not *Bob*, since the latter is played by the intruder) and two for the second session as follows:

- P1: Session 1, Role Alice ($S1\_Alice.Actor = a$, $S1\_Alice.B = i$)
- P2: Session 2, Role Alice ($S2\_Alice.Actor = a$, $S2\_Alice.B = b$)
- P3: Session 2, Role Bob ($S2\_Bob.A = a$, $S2\_Bob.Actor = b$)

For the sake of simplicity, in the remainder of this section we focus on interpolants that either constrain the intruder knowledge or the instantiation of other variables, but nothing prevents an interpolant from combining the two types of constraints.

Interpolants constraining the intruder knowledge.. We illustrate this type of interpolant by considering the execution path of the NSL running example detailed in Table 4. The execution path is the one given in Example 9 (and in Table 1) but it focuses on P2 and P3 for readability. As we already discussed in Section 2, the running example is secure against a MITM attack. Therefore, when SPiM reaches the goal location (with ID = 7 in Table 4), the state is unsatisfiable and, by using the *Learn* rule, SPiM produces an interpolant (annotation) and propagates it back using the *Conjoin* rule. The interpolant generated in location ID 6 and reported in Figure 13-left constrains the intruder knowledge (*IK*) listing which messages have to be in *IK* and which messages must not be in *IK*. We note that the interpolant in Figure 13-left contains only constants and no variables. This is due to the implementation of *IK* in SPiM.

$$
\begin{array}{rl}
\neg IK \vdash \{na,a\} \wedge [& \\
IK \vdash \{na,na,a\} \vee & \\
IK \vdash \{na,nb,a\} \vee & \\
[IK \vdash \{na,a\}_{pk(a)} \wedge IK \vdash inv(pk(a))] \vee & \neg IK \vdash \{na,a\} \wedge \\
[IK \vdash \{na,a\}_{pk(i)} \wedge IK \vdash inv(pk(i))] \vee & IK \vdash \{na,a\}_{pk(i)} \wedge \\
[IK \vdash \{na,a\}_{pk(b)} \wedge IK \vdash inv(pk(b))] \vee & IK \vdash inv(pk(i)) \\
[IK \vdash \{na\} \wedge IK \vdash a]] &
\end{array}
$$

Fig. 13. Two examples of interpolants constraining the intruder knowledge

In fact, IK contains only constants since we only store the actual value (constant) of each component of a message sent to the intruder.

When we reach the end of the execution of the protocol (i.e., location with ID $= 6$) with the constraint $S2\_Bob.A \neq i$, then the authentication property (ID $= 7$) cannot be reached. Therefore, it is *not* possible for the intruder to craft the message *S2_Bob.Nb* encrypted with the public key of *S2_Bob.B* without playing the role of the agent *S2_Bob.A*. This is due to the impossibility for the intruder to obtain the nonces *S2_Bob.Na* or *S2_Bob.Nb* without playing the role of *S2_Bob.A*, i.e., without decrypting a message containing *S2_Bob.Na* or *S2_Bob.Nb*.

The annotation (of location 6 in Table 4) in Figure 13-left[9], in fact, states that it is impossible to reach the goal location if the intruder does not know the message $\{na,a\}$ and (at the same time) one of the followings holds:

- $IK \vdash \{na,na,a\} \vee IK \vdash \{na,nb,a\}$. In fact, the only way for the intruder to know one of these two messages is to play the role of a, which contradicts the goal.
- $IK \vdash \{na,a\}_{pk(*)} \wedge IK \vdash inv(pk(*))$, where $*$ refers to one of the two honest agents or the intruder. In fact, if the intruder knows the inverse key to decrypt this message, he has to either play the role of a or b. The former contradicts the first conjunct of the goal, the latter contradicts the second.
- $IK \vdash na \wedge IK \vdash a$. This constraint (together with the initial $\neg IK \vdash \{na,a\}$) states that if the intruder knows the components of $\{na,a\}$ but has not been able to pair na and a, then he will not reach the goal location.

Another similar example is reported in Figure 13-right. The annotation has been generated by the *Conjoin* rule for the location between the actions with ID 4 and 5 (Table 4) and shows how the previous annotation (Figure 13-left) simplifies during the backtrack phase.

Interpolants constraining instantiation of variables.. The second type of interpolants is the one constraining the instantiation of program variables. For example, the following interpolant constraints the instantiation of the agent's variables in such a way that a path where the intruder plays the role of Alice will not be (re-)executed since it is (trivially) in contrast with the authentication goal:

$$S2\_Bob.A = i$$

[9]Note that, for the sake of readability, we refer to the constants of the nonces using the notation na,nb instead of c_0,c_1.

6. Related work

To the best of our knowledge, there is no other tool for security protocol analysis that uses a speed-up technique based on Craig's interpolation. We now discuss some further related work on interpolation, in addition to the works we already considered in detail in the other sections of the paper.

In [15], McMillan presented the IntraLA algorithm that we have used as a basis for this work. However, our application field is network security whereas IntraLA has been developed for software verification, and this has led to a number of substantial differences between the two works. First of all, our case studies are security protocols, and thus parallel programs, whereas IntraLA works on sequential ones. For this reason, we have defined a simple programming language (SiL) with some protocol-oriented features and provided a translation procedure from protocol specifications (expressed in ASLan++) into SiL programs (proving the correctness of the translation with respect to the semantics of ASLan++). In particular, given the object of our study, SiL allows one to express statements aimed at handling the actions of the DY intruder. The DY theory has then been used both in the symbolic execution of a program graph (*Decide* rule, Section 4) and for interpolants generation (*Learn* rule, Section 4). The nature of the goals that we verify also differ from the ones in [15], as they are directly related to security goals like authentication and confidentiality. The same differences can be found between SPiM and IMPACT II (the implementation of [15]): IMPACT II takes as input control flow graphs from C programs and has been tested on the source codes of drivers. The algorithm implementations do also have some differences. In particular, in SPiA, we have implemented an optimization according to which an interpolant is calculated, at a given node or edge, only when the graph presents an unexplored path that can be blocked by such an interpolant.

Recently, McMillan has proposed in [17] a variation of IntraLA that mainly adapts IntraLA to large-block encoding (LBE). This technique reduces the abstract reachability tree used by the IntraLA algorithm, for example by simplifying the tree produced from very long sequences of if statements. Moving from original trees to the ones produced with LBE is not a trivial task and requires further investigation. Introducing LBE could speed up our tool too but, as we have already discussed in Section 5.1, we implemented SPiM mainly to show that interpolation can concretely be used as a speed-up technique together with the DY intruder model in the context of security protocols. Other works by McMillan that exploit the use of Craig interpolation in model checking are [32, 33], but interpolants are used there in a different way, i.e., to apply interpolant-based image approximation.

Besides McMillan's works on interpolation applied to model checking, there are a number of model checkers that implement different techniques to speed-up the search for goal locations. In particular, for the purpose of the comparison with SPiM and in addition to the tools already considered in Section 5.1, we consider here four security protocol analysis tools that implement the DY intruder theory: Maude-NPA [34], ProVerif [4], Scyther [35] and Tamarin [36].

Besides DY, Maude-NPA supports a wide range of theories such as the "associative-commutative plus identity" theory. Maude-NPA has been implemented with particular focus on performances and in fact, during the analysis, it takes advantage of various state-space reduction techniques. These range from a modified version of the lazy intruder (called "super lazy intruder") to a partial-order reduction technique. The ideas behind the speed-up techniques of Maude-NPA are very similar to the ones of SPiM: reduce the number of states to explore and try to not explore a state after having the evidence that from this state the model checker will never reach the goal location (i.e., will never reach the initial state given that Maude-NPA performs a backward reachability search). As for all the back-ends of the AVANTSSAR Platform (discussed in Section 5.1), in principle we do not see any incompatibility in

combining the interpolation-based technique we have proposed in this paper with the speed-up techniques implemented in Maude-NPA. However, Maude-NPA performs backward reachability analysis whereas our technique has been defined for forward reachability analysis. This does not prevent possible useful interaction between the two approaches but it might require a non-trivial adaptation of the interpolation-based algorithm.

In ProVerif, security protocols are represented using Prolog rules in order to handle multiple executions. It implements an efficient algorithm that, combined with a unification technique along with rule optimization procedures, handles the problem of state-space explosion. Due to the particular nature of the techniques it implements, it is not clear if ProVerif could further improve its performance by integrating an interpolation-based technique.

Scyther uses a pattern-refinement algorithm that provides concise representations of (infinite) sets of traces. It does not use approximation methods nor abstraction techniques and it could thus benefit from including our technique, in particular, when unbounded verification is performed. However, as with Maude-NPA, due to Scyther's backward searching algorithm, this integration would require further study.

Tamarin uses a constraint-solving algorithm and a symbolic representation of states like SPiM, but supports analysis for an unbounded number of protocol sessions. Intruder capabilities and protocols are specified jointly as a set of (labeled) multiset rewriting rules. Tamarin is particularly well suited for the analysis of protocols that use the Diffie-Hellman key exchange, which SPiM does not handle. One of the main difficulties one might have in implementing our speed-up technique in Tamarin is thus with the Diffie-Hellman key representation. However, since Tamarin uses a (labeled) operational semantics that is similar to the one used in SPiM, it might still be feasible to adapt the interpolation technique successfully.

7. Concluding remarks

We believe that our interpolation-based method, together with its prototype implementation in the SPiM tool and our experimental evaluation, shows that we can indeed use interpolation to reduce the search space and speed up the execution also in the case of security protocol verification. In particular, as we have shown, we can use a standard security protocol specification language (ASLan++, but, we believe that with little effort, also other languages that specify the different protocol roles as interacting processes could be used) and translate automatically into SPiM's input language SiL with the guarantee that in doing so we will not introduce nor lose any attack. The tool then proceeds automatically and concludes reporting either all the different reachable states (from which one or more abstract attack traces can be extracted) or that no attack has been found for the given specification.

As future work, we plan to increment our experimental results by considering further (and more complex) security protocols, such as those described in [37] and in the standard literature. This will allow us to collect further evidence as to what extent interpolation can indeed increase the performance of SPiM.

More importantly, as we remarked above, we are not aware of any other tool for security protocol verification that uses an interpolation-based speed-up technique, and we believe that actually interpolation might be proficiently used in addition (and not in alternative) to other optimization techniques for security protocol verification. We are thus currently investigating possible useful interactions between interpolation and such optimization techniques, given that there are no theoretical or technical incompatibilities between them. This will allow us to enhance SPiM and promote its performance closer to the

level of the more mature tools. Symmetrically, it would be interesting to investigate also whether such mature tools might benefit from the integration of interpolation-based techniques such as ours to provide an additional boost to their performance. This will of course be a much more challenging endeavor to undertake, as it will possibly require some internal changes to already deployed tools, but given our close scientific relations to some of the tool developers, we are hopeful that we will be able to carry out some attempts in this direction.

Appendix A. ASLan++ specification of NSL.

```
specification NSL
channel_model CCM

entity Environment {
  symbols
   a,b:agent;

  entity Session (A, B: agent) {

    entity Alice (Actor, B: agent) {

      symbols
        Na, Nb: text;

      body {
        Na := fresh();
        Actor  ->  B: {Na.Actor}_pk(B);
        B  ->  Actor: {Na.?Nb.B}_pk(Actor);
        Actor  ->  B: {auth:(Nb)}_pk(B));
      }
    }

    entity Bob (A, Actor: agent) {

      symbols
        Na, Nb: text;

      body {
        ? -> Actor: {?Na.?A}_pk(Actor);
        Nb := fresh();
        Actor  ->  A: {Na.Nb.Actor}_pk(A);
        A  ->  Actor: {auth:(Nb)}_pk(Actor);
      }
    }

    body { % of Session
      new Alice(A,B);
      new Bob(A,B);
    }

    goals
     auth:(_) A *-> B;
  }

  body { % of Environment
    any Session(a,i);
    any Session(a,b);
```

```
48   }
49 }
```

Appendix B. Proof of Lemma 1

Proof. We show two representative cases; the other ones can be treated similarly. (i) Let the statement I considered be the receipt of a message having the form:

```
1 entity Environment {
2 ...
3  entity Session (A, B: agent) {
4   ...
5   entity Alice(Actor, B: agent) {
6    ...
7    body {
8     ...
9     B  ->  Actor: M(?A_1,...?A_n)
10    ...
11 }
```

The corresponding ASLan rule r has the form:

```
1 step ...(...) :=
2   PF'.
3   iknows(M'(N_1,..., N_n)).
4   state_Alice(B_1,...,B_m)
5   =>
6   R'.
7   state_Alice(B'_1,...,B'_m)
```

where M' is the ASLan translation of M, $n \leqslant m$ and $\forall j.1 \leqslant j \leqslant m$ if $j = f(Alice,A\_i)$ for some $1 \leqslant i \leqslant n$, then $B'\_f(Alice,A\_i) = N\_i$, otherwise $B'\_j = B\_j$.
For simplicity, we ignore in the variable names the prefixes referring to the session instance. w has the form:

```
1 if (IK |- M''(Y_1,...,Y_n))
2   then
3    Alice.A_1 = Y_1;
4    ...
5    Alice.A_N = Y_N;
6   else
7    end
```

where M'' is the SiL translation of M where we have replaced $?A\_1,...,?A\_n$ with $Y\_1,...,Y\_n$.
$(\Rightarrow)$ Let S' be such that $S \xrightarrow{r} S'$. By the semantics of ASLan, there must exist a substitution σ such that:

$$iknows(M'(N\_1,...,N\_n)).state\_Alice(B\_1,...,B\_m)\sigma \subseteq \lceil S \rceil^H$$

Furthermore, there exists a substitution σ'' such that:

$$state\_Alice(B'\_1,...,B'\_m)\sigma\sigma'' \subseteq \lceil S' \rceil^H$$

Then we can build an extension $\bar{\varsigma}$ of ς such that:

- $\bar{\varsigma}(Y\_i) = \sigma(N\_i)$ for $1 \leqslant i \leqslant n$;
- $\bar{\varsigma}(A) = \sigma(A)$ for any other variable A.

It follows that $M'(N\_1,...,N\_n)\sigma \sim M''(Y\_1,...,Y\_n)\bar{\varsigma}$ and since $iknows(M'(N\_1,...,N\_n))\sigma \subseteq \lceil S \rceil^H$ then, by hypothesis, $M''(Y\_1,...,Y\_n) \in DY(\bar{\varsigma}(IK))$ which implies $< IK \vdash M''(Y\_1,...,Y\_n), \bar{\varsigma} >\Downarrow true$. By using this fact in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{< Y\_1, \bar{\varsigma} >\Downarrow \bar{\varsigma}(Y)} \quad \vdots}{< \Phi_1, \bar{\varsigma} >\Downarrow \varsigma_1} \quad \dfrac{}{< Alice.A\_2 := Y\_2, \varsigma_1 >\Downarrow \varsigma_2}}{< \Phi_2, \bar{\varsigma} >\Downarrow \varsigma_2} \quad \cdots \quad \dfrac{\dfrac{}{< Y_n, \varsigma_{n-1} >\Downarrow \bar{\varsigma}(Y_n)}}{< \Psi_n, \varsigma_{n-1} >\Downarrow \varsigma_n \equiv \varsigma'}}{< \Phi_{n-1}, \bar{\varsigma} >\Downarrow \Sigma_{n-1}} \quad \vdots}{< \Phi_n, \bar{\varsigma} >\Downarrow \varsigma'}$$

$$\dfrac{< IK \vdash M''(Y\_1,...,Y\_n), \bar{\varsigma} >\Downarrow true \qquad < \Phi_n, \bar{\varsigma} >\Downarrow \varsigma'}{< if\ (IK \vdash M''(Y\_1,...,Y\_n))\ then\ \Phi_n\ else\ end, \bar{\varsigma} >\Downarrow \varsigma'}$$

we get that $< w, \bar{\varsigma} >\Downarrow \varsigma' \equiv \varsigma_n$, where we have used the abbreviations:

$$\Phi_i \equiv Alice.A\_1 := Y\_1; \ldots; Alice.A\_i := Y\_i;$$

$$\Psi_i \equiv Alice.A\_1 := Y\_i; \ldots; Alice.A\_i := Y\_n;$$

$$\varsigma_i \equiv \bar{\varsigma}[Alice.A\_1 \leftarrow \bar{\varsigma}(Y\_1), ..., Alice.A\_i \leftarrow \bar{\varsigma}(Y\_i)].$$

We have that $S'(Alice, f(Alice,A\_i)) = \sigma(B'\_f(Alice,A\_i)) = \sigma(N\_i) = \varsigma'(Alice.A\_i)$, for $1 \leqslant i \leqslant n$. Since S' and ς' coincide with S and ς, respectively, for what concerns the other variables, we can conclude $S' \sim \varsigma'$.

($\Leftarrow$) Assume there exists an extension $\bar{\varsigma}$ of ς such that $< w, \bar{\varsigma} >\Downarrow \varsigma'$. The case when $< IK \vdash M''(Y\_1,...,Y\_n),\ \bar{\varsigma} >\Downarrow false$ is trivial, since $\varsigma' \equiv \bar{\varsigma}$ and we can easily take $S' \equiv S$. Let $< IK \vdash M''(Y\_1,...,Y\_n), \bar{\varsigma} >\Downarrow true$. Then $M''(Y\_1,...,Y\_n)\bar{\varsigma} \in DY(\varsigma(IK))$. It follows that we can choose a substitution σ such that $\sigma(N\_i) = \bar{\varsigma}(Y\_i)$, for $1 \leqslant i \leqslant n$, and thus $iknows(M'(N\_1,...,N\_n))\sigma \subseteq \lceil S \rceil^H$. By applying the rules of SiL semantics as above and the rule r, we get an S' such that $S \xrightarrow{r} S'$ and $S' \sim \varsigma'$.

(ii) Let us assume that Alice wants to authenticate Bob and consider, without loss of generality, a program instance pi where $pi(Alice) = a$ and $pi(Bob) = i$, since if *Bob* is played by an honest agent, then the authentication property is trivially satisfied. I has the form:

```
1 entity Environment {
2  ...
3  entity Session (A, B: agent) {
4  ...
5   entity Alice(Actor, B: agent) {
6   ...
7    body {
8     ...
9     B -> Actor: auth:(M);
10    ...
11   }
12  ...
13  }
```

```
14   ...
15   goals
16    auth:(_) B *-> A;
17    ....
18 }
```

and is a particular case of a receipt. As such, it is translated as a common receipt, treated in case (i), plus special constructs/rules aimed at handling the goal conditions, which will be treated here. The corresponding ASLan attack state is described by:

```
1 attack_state auth(M', Actor, B,..) :=
2   not(dishonest(B)).
3   not(witness(B, Actor, M', auth)).
4   request(Actor, B, M', auth, ...)
```

where M' is the ASLan translation of M (for simplicity, we assume here that the payload on which authentication is based is the whole message). We also add a corresponding ASLan rule r of the form:

```
1   AS => AS.attack
```

which simply adds the 0-ary predicate *attack* to an attack state AS containing the predicates described above.
The corresponding SiL statement w has the form:

```
1 if(not(Alice.B = i) and not(witness(Alice.B, Alice.Actor, M'')))
2  then
3   attack := true;
4  else
5   skip;
```

where M'' is the SiL translation of M. First, we notice that while the rule r can be applied at any step in an ASLan run, the corresponding SiL statement w is placed, by the translation procedure, immediately after the receipt instruction. For simplicity, we will restrict to consider those ASLan runs where attack rules concerning authentication goals, like r above, are only applied immediately after the receipt of the corresponding message. This can be done without loss of generality (and is also the reason why we do not need a *request* predicate in SiL).
$(\Rightarrow)$ In order to apply the rule r, by the semantics of ASLan, there must exist a substitution σ such that:

$$request(Actor,B,M',auth,\ldots).state\_Alice(\ldots,B,\ldots)\sigma \subseteq \lceil S \rceil^H$$

where, in particular, $\sigma(B) = S(Alice, f(Alice,B))$. At the same time, we have: $dishonest(B)\sigma \not\subseteq \lceil S \rceil^H$ and $witness(B,Actor,M',auth)\sigma \not\subseteq \lceil S \rceil^H$.

Since, as for every ASLan state, $dishonest(i) \subseteq \lceil S \rceil^H$, we get $\sigma(B) \neq i$. Let $\bar{\varsigma}$ be an extension of ς. By hypothesis, $S \sim \varsigma$, from which we infer $\sigma(B) = S(Alice, f(Alice,B)) = \varsigma(Alice.B) = \bar{\varsigma}(Alice.B) \neq i$. With analogous arguments, we infer $(\bar{\varsigma}(Alice.B), \bar{\varsigma}(Alice.Actor), \bar{\varsigma}(M'')) \notin \bar{\varsigma}(witness)$. By using these facts, we obtain the derivation in Figure 14.

We have that S' and ς' differ from S and $\bar{\varsigma}$, respectively, only for the value of the predicate *attack*. By observing that $attack \subseteq \lceil S' \rceil^H$ and $\varsigma'(attack) = true$, we conclude $S' \sim \varsigma'$.
$(\Leftarrow)$ Let $\bar{\varsigma}$ be an extension of ς such that $< w, \varsigma > \Downarrow \varsigma'$. The case when $< \Psi, \bar{\varsigma} > \Downarrow false$, where Ψ is defined as in $(\Rightarrow)$ above, is trivial. Let us consider $< \Psi, \bar{\varsigma} > \Downarrow true$. By hypothesis, $S \sim \varsigma$ and thus the

preconditions of r concerning the predicates *dishonest* and *witness* are enabled in S. As for the condition on the *request*, it is enabled by the fact that the corresponding receipt has just been encountered, by construction of a SiL graph. It follows that r can be applied and we get an ASLan state S', which differs from S only in the fact that $attack \subseteq \lceil S' \rceil^H$. Moreover, by applying the same derivation as in case ($\Rightarrow$) above, we have $\varsigma'(attack) = true$, from which we conclude $S' \sim \varsigma'$. $\square$

References

[1] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S.E. Ponta, M. Rocchetto, M. Rusinowitch, M. Torabi Dashti, M. Turuani and L. Viganò, The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures, in: *TACAS*, C. Flanagan and B. König, eds, LNCS 7214, Springer, 2012, pp. 267–282. doi:10.1007/978-3-642-28756-5_19. http://www.avantssar.eu.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò and L. Vigneron, The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications, in: *CAV*, LNCS 3576, Springer, 2005, pp. 281–285. doi:10.1007/11513988_27. http://www.avispa-project.org.

[3] D. Basin, S. Mödersheim and L. Viganò, OFMC: A symbolic model checker for security protocols, *International Journal of Information Security* **4**(3) (2005), 181–208. doi:10.1007/s10207-004-0055-7.

[4] B. Blanchet, An Efficient Cryptographic Protocol Verifier Based on Prolog Rules, in: *CSFW*, IEEE CS, 2001, pp. 82–96.

[5] C. Cremers and S. Mauw, *Operational Semantics and Verification of Security Protocols*, Springer, 2012.

[6] S. Escobar, C. Meadows and J. Meseguer, Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties, in: *FOSAD*, Springer, 2007, pp. 1–50.

[7] G. Lowe, Breaking and Fixing the Needham-Shroeder Public-Key Protocol Using FDR, in: *TACAS*, LNCS 1055, Springer, 1996, pp. 147–166.

[8] S. Mödersheim and L. Viganò, The Open-Source Fixed-Point Model Checker for Symbolic Analysis of Security Protocols, in: *FOSAD 2008/2009*, LNCS 5705, Springer, 2009, pp. 166–194. doi:10.1007/978-3-642-03829-7_6.

[9] M. Rusinowitch and M. Turuani, Protocol insecurity with a finite number of sessions and composed keys is NP-complete, *Theor. Comput. Sci.* **299**(1–3) (2003), 451–475, ISSN 0304-3975.

[10] A. Armando, R. Carbone, L. Compagna, J. Cuéllar and L. Tobarra Abad, Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps, in: *FMSE*, ACM Press, 2008.

[11] A. Armando, G. Pellegrino, R. Carbone, A. Merlo and D. Balzarotti, From Model-Checking to Automated Testing of Security Protocols: Bridging the Gap, in: *TAP*, LNCS 7305, Springer, 2012, pp. 3–18.

[12] L. Viganò, The SPaCIoS Project: Secure Provision and Consumption in the Internet of Services, in: *ICST*, IEEE CS Press, 2013. doi:10.1109/ICST.2013.75. www.spacios.eu.

[13] M. Büchler, J. Oudinet and A. Pretschner, Security Mutants for Property-Based Testing, in: *TAP*, LNCS 6706, Springer, 2011, pp. 69–77.

[14] K.L. McMillan, Applications of Craig Interpolants in Model Checking, in: *TACAS*, LNCS 3440, Springer, 2005, pp. 1–12. ISBN 978-3-540-25333-4.

[15] K.L. McMillan, Lazy Annotation for Program Testing and Verification, in: *CAV*, LNCS 6174, Springer, 2010, pp. 104–118. ISBN 978-3-642-14294-9.

[16] K.L. McMillan, An interpolating theorem prover, *Theoretical Computer Science* **345**(1) (2005), 101–121, ISSN 0304-3975.

[17] K.L. McMillan, Lazy Annotation Revisited, in: *Computer Aided Verification - 26th International Conference*, 2014, pp. 243–259.

[18] D. Dolev and A. Yao, On the Security of Public-Key Protocols, *IEEE Transactions on Information Theory* **2**(29) (1983).

[19] W. Craig, Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory, *The Journal of Symbolic Logic* **22**(3) (1957), 269–285, ISSN 00224812.

[20] J.C. King, Symbolic execution and program testing, *CACM* **19**(7) (1976), 385–394, ISSN 0001-0782.

[21] M. Turuani, The CL-Atse Protocol Analyser, in: *Term Rewriting and Applications*, LNCS 4098, Springer, 2006, pp. 277–286. ISBN 978-3-540-36834-2.

[22] A. Armando and L. Compagna, SATMC: a SAT-based Model Checker for Security Protocols, in: *JELIA*, LNAI 3229, Springer, 2004, pp. 730–733.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{<Alice.B,\bar{\varsigma}>\Downarrow \bar{\varsigma}(Alice.B)} \quad \dfrac{}{<i,\bar{\varsigma}>\Downarrow i}}{<Alice.B=i>\bar{\varsigma}>\Downarrow false}}{<not(Alice.B=i),\bar{\varsigma}>\Downarrow true} \quad \dfrac{\dfrac{\dfrac{}{<Alice.B,\bar{\varsigma}>\Downarrow \bar{\varsigma}(Alice.B)} \quad \dfrac{}{<Alice.Actor,\bar{\varsigma}>\Downarrow \bar{\varsigma}(Alice.Actor)} \quad \dfrac{}{<M'',\bar{\varsigma}>\Downarrow \bar{\varsigma}(M'')}}{<\Phi,\bar{\varsigma}>\Downarrow false}}{<not(\Phi),\bar{\varsigma}>\Downarrow true}}{<\Psi,\bar{\varsigma}>\Downarrow true} \quad \dfrac{\dfrac{}{<true,\bar{\varsigma}>\Downarrow true}}{<attack:=true,\bar{\varsigma}>\Downarrow \bar{\varsigma}[true/attack]}}{<if\ \Psi\ then\ attack:=true\ else\ skip,\bar{\varsigma}>\Downarrow \bar{\varsigma}[true/attack]\equiv\varsigma'}$$

In the derivation, we used $\Phi \equiv witness(Alice.B, Alice.Actor, M'')$ and $\Psi \equiv not(Alice.B = i)\ and\ (not(\Phi))$ as abbreviations.

Fig. 14.: A derivation for an authentication goal checking by the operational semantics of SiL.

[23] M. Rocchetto, L. Viganò, M. Volpe and G. Dalle Vedove, Using Interpolation for the Verification of Security Protocols, in: *STM*, Springer, 2013, pp. 99–114. doi:10.1007/978-3-642-41098-7_7.
[24] S. Mödersheim and L. Viganò, Secure Pseudonymous Channels, in: *Esorics*, LNCS 5789, Springer, 2009, pp. 337–354. doi:10.1007/978-3-642-04444-1_21.
[25] V. Cortier, S. Delaune and P. Lafourcade, A Survey of Algebraic Properties used in Cryptographic Protocols, *Journal of Computer Security* **1** (2006), 1–43.
[26] S. Mödersheim, Algebraic Properties in Alice and Bob Notation, in: *Ares*, IEEE CS, 2009, pp. 433–440.
[27] D. von Oheimb and S. Mödersheim, ASLan++ — A formal security specification language for distributed systems, in: *FMCO*, LNCS 6957, Springer, 2010, pp. 1–22.
[28] AVANTSSAR, Deliverable 2.3 (update): ASLan++ specification and tutorial, 2011, Available at http://www.avantssar.eu.
[29] G. Kahn, Natural Semantics, in: *STACS, 4th Annual Symposium*, 1987, pp. 22–39. doi:10.1007/BFb0039592. http://dx.doi.org/10.1007/BFb0039592.
[30] L. de Moura and N. Bjorner, Z3: An Efficient SMT Solver, in: *TACAS*, LNCS 4963, Springer, 2008, pp. 337–340. ISBN 978-3-540-78799-0.
[31] K.L. McMillan, Interpolants from Z3 proofs, in: *FMCAD*, 2011, pp. 19–27. ISBN 978-0-9835678-1-3.
[32] K.L. McMillan, An interpolating theorem prover, in: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2004, pp. 16–30.
[33] K.L. McMillan, Lazy Abstraction with Interpolants, in: *CAV, 18th International Conference*, Springer, 2006, pp. 123–136.
[34] S. Escobar, C. Meadows, J. Meseguer and S. Santiago, State space reduction in the Maude-NRL protocol analyzer, *Information and Computation* **238** (2014), 157–186.
[35] C.J.F. Cremers, The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols, in: *Proceedings of CAV 2008*, LNCS 5123, Springer, 2008, pp. 414–418.
[36] B. Schmidt, R. Sasse, C. Cremers and D.A. Basin, Automated Verification of Group Key Agreement Protocols, in: *IEEE Symposium on Security and Privacy, SP*, 2014, pp. 179–194. doi:10.1109/SP.2014.19.
[37] C. Boyd and A. Mathuria, *Protocols for Authentication and Key Establishment*, Springer, 2010.